

E.T.S. de Ingeniería Industrial,  
Informática y de Telecomunicación

# RECONOCIMIENTO FACIAL Y EXTRACCIÓN DE INFORMACIÓN MEDIANTE DEEP LEARNING A PARTIR DE DATOS OBTENIDOS EN RED SOCIAL



Grado en Ingeniería Informática

Trabajo Fin de Grado

Alumno: Peio García Pinilla

Director: Mikel Galar Idoate

Pamplona, 30 de abril de 2020

# Agradecimientos

Quiero agradecer a todas aquellas personas que han contribuido a la realización de este proyecto:

- Primero, quiero agradecer a mi tutor de la Universidad, Mikel Galar, por la paciencia e interés que ha mostrado sobre este proyecto desde el primer momento.
- Segundo, a todos aquellos amigos y amigas que se han ofrecido a colaborar prestando sus datos para realizar las pruebas, y que aparecen a lo largo de toda la memoria.
- Finalmente, quiero agradecer de la forma más especial posible a mi madre y a mi padre, ya que, sin ellos, no hubiese sido posible llegar hasta aquí y realizar el proyecto.

Gracias a todas ellas.

Peio

# Resumen

En plena era de la información donde el Big Data se muestra en auge, y las Redes Sociales forman parte ya de nuestras vidas, diariamente compartimos datos de nuestro día a día sin ser conscientes de la cantidad de información que puede ser extraída de ellos.

Este proyecto consiste en crear un sistema de reconocimiento facial a través de técnicas propias del aprendizaje profundo o *Deep Learning*, en donde los datos que empleamos para su aprendizaje son obtenidos a partir de la red social Instagram, en este caso, las imágenes.

Una vez reconocido, se realizará un proceso de minería de datos extrayendo información como pueden ser gustos, aficiones o tendencias del usuario en cuestión, según las fotos publicadas en su perfil.

## Palabras clave

- Reconocimiento facial
- Aprendizaje profundo
- Red neuronal
- Instagram
- Big Data

## Summary

In the era of information where Big Data is booming, and Social Networks are already part of our lives, daily we share data from our day to day without being aware of the amount of information that can be extracted from them.

This project consists of creating a facial recognition system through techniques of Deep Learning, where the data we use for their learning are obtained from the Instagram social network, in this case, the images.

Once recognized, a data mining process will be carried out by extracting information such as tastes, hobbies or trends of the user in question according to the photos they have shared in their profile.

## Key words

- Face recognition
- Deep Learning
- Neural Network
- Instagram
- Big Data

# Laburpena

Big Data etengabe hazten ari den eta sare sozialak gure bizitzaren zati diren aroan bizi gara, informazioaren aroan. Uneoro gure eguneroko datuak partekatzen ditugu haietatik atera daitekeen informazio guztiaz ohartu gabe.

Proiektu hau aurpegiko errekonozimendu sistema bat sortzean datza. Bertan, sakoneko ikasketa edo Deep Learning tekniken bitartez, ikasteko erabiltzen ditugun datuak Instagram sare sozialetik lortzen dira, kasu honetan irudiak.

Behin aurpegia ezagututa, datuen meatze prozesua egingo da, erabiltzailearen zaletasunak, ohiturak edota joerak azaleratuz, ezagututako pertsonaren profilean partekatutako argazkien arabera.

## Hitz klabeak

- Aurpegi errekonozimendua
- Ikaskuntza sakona
- Neurona-sarea
- Instagram
- Big Data

# Índice de Contenido

<b>1- INTRODUCCIÓN Y MOTIVACIÓN .....</b>	<b>11</b>
1.1 ENTORNO DE TRABAJO .....	12
<b>2- PRELIMINARES .....</b>	<b>13</b>
2.1 MACHINE LEARNING .....	13
2.2 DEEP LEARNING .....	14
2.2.1 ¿Por qué el Deep Learning está creciendo?.....	15
2.2.2 Proceso iterativo.....	16
2.2.3 Redes Neuronales .....	17
Introducción .....	17
Funciones de activación .....	18
Descenso por gradiente y actualización de pesos .....	19
2.2.4 Redes Neuronales Profundas.....	22
Introducción .....	22
Forward y backward propagation .....	23
Parámetros – Hiperparámetros.....	24
2.2.5 Optimización del modelo.....	24
Bias – Varianza .....	24
Regularización .....	25
Normalización.....	29
Algoritmos de optimización .....	31
Hyperparameter tuning.....	34
Softmax layer.....	35
Receta básica para Machine Learning.....	36
2.2.6 Redes Neuronales Convolucionales.....	37
Introducción .....	37
Estructura y funcionamiento de las CNN .....	38
Ejemplos de arquitecturas clásicas.....	41
ResNets y Redes Inception .....	42
Data augmentation .....	42
Reconocimiento facial .....	43
2.3 PREÁMBULO .....	44
<b>3- OBTENCIÓN DE DATOS EN BRUTO .....</b>	<b>46</b>
3.1 BASE DE DATOS .....	46
3.2 WEB SCRAPING.....	49
3.2.1 Legalidad .....	49
3.2.2 Proceso de extracción.....	50
<b>4- GENERACIÓN DE LA BASE DE DATOS DE CARAS .....</b>	<b>52</b>
4.1 DETECCIÓN DE ROSTRO .....	52
4.1.1 Clasificador Haar .....	53
4.1.2 Implementación del detector .....	54
4.2 CLASIFICACIÓN DE CARAS .....	58

4.2.1	Procesamiento de imágenes .....	58
	CNN y generación de embeddings .....	58
4.2.2	PCA (Principal Component Analysis) .....	61
4.2.3	Reconociendo la cara del usuario, Algoritmo K-means.....	62
	Funcionamiento de K-means.....	62
	Implementación del algoritmo.....	64
4.2.4	Aumentando el conjunto de datos de entrenamiento (Data Augmentation) .....	66
<b>5-</b>	<b>EXTRACCIÓN DE LA INFORMACIÓN ADICIONAL: PERFILANDO A LOS USUARIOS .....</b>	<b>70</b>
5.1	IMAGE CAPTIONING .....	71
5.1.1	Implementación del modelo.....	73
	Dataset .....	74
	Preparación de dataset de fotos .....	74
	Preparación del dataset de descripciones.....	76
	Imágenes similares .....	78
	Modelo LSTM .....	80
	Entrenamiento del modelo .....	81
	Generación de nuevas descripciones .....	82
	Resultados y pruebas .....	82
5.2	TENDENCIAS DEL USUARIO.....	84
5.2.1	Modelo Bag of Words.....	84
5.2.2	Palabras relevantes en las descripciones .....	84
	Implementación .....	84
5.3	EXTRACCIÓN DE INFORMACIÓN ADICIONAL DEL USUARIO .....	87
5.3.1	Popularidad .....	87
5.3.2	Usuarios frecuentes.....	88
5.3.3	Tags o tendencias reales .....	88
5.3.4	Emojis más usados .....	88
5.3.5	Títulos de las publicaciones .....	89
5.3.6	Comentarios .....	89
<b>6-</b>	<b>RECONOCIMIENTO FACIAL.....</b>	<b>90</b>
6.1	RECONOCIMIENTO A TIEMPO REAL .....	90
6.2	CLASIFICACIÓN .....	91
6.3	RESULTADOS Y GRÁFICAS DE RENDIMIENTO .....	92
<b>7-</b>	<b>CONCLUSIONES .....</b>	<b>97</b>
<b>8-</b>	<b>LÍNEAS FUTURAS .....</b>	<b>98</b>
8.1	MODELO BIG FIVE .....	98
8.2	APLICACIÓN MÓVIL.....	99
8.3	MEJORAR EFICIENCIA DEL PROCESO.....	99
8.4	AUMENTAR ALCANCE.....	100
8.5	TRASLADO A LA PRÁCTICA.....	100
<b>9-</b>	<b>BIBLIOGRAFÍA .....</b>	<b>101</b>

# Índice de ilustraciones

Figura 1. Gráfico del rendimiento de modelos según cantidad de datos empleada.	
Fuente: researchgate .....	15
Figura 2. Proceso de aprendizaje iterativo. Fuente: medium.....	16
Figura 3. Representación visual de un perceptrón. Fuente: Alberto Torres .....	18
Figura 4. Funciones de activación más comunes. Fuente: medium .....	19
Figura 5. Representación gráfica de la función de coste. Fuente: IArtificial.....	20
Figura 6. Iteraciones del gradiente. Fuente: cs.eu .....	21
Figura 7. Esquema de una red multicapa. Fuente: ResearchGate .....	22
Figura 8. Representación por bloques. Fuente: Medium.....	23
Figura 9. Bias y varianza. Fuente: lokad .....	24
Figura 10. Modelo sobre-entrenado. Fuente: Wikipedia .....	25
Figura 11. Función de coste con regularización. Fuente: Coursera.....	25
Figura 12. Regularización L1. Fuente: Coursera. ....	26
Figura 13. Actualización de pesos con regularización. Fuente: Coursera.....	26
Figura 14. Función de tangente hiperbólica y regularización.....	26
Figura 15. Regularización por abandono. Fuente: Tech-Quantum .....	27
Figura 16. Regularización por parada temprana. Fuente: Medium.....	28
Figura 17. Coste no-normalizado vs. normalizado. Fuente: Coursera. ....	29
Figura 18. Media aritmética del conjunto de entrenamiento. Fuente: Coursera. ....	30
Figura 19. Desviación típica del conjunto de datos. Fuente: Coursera.....	30
Figura 20. Pasos de la Normalización. Fuente: Coursera .....	30
Figura 21. Expresión del proceso de Normalización. Fuente: Coursera. ....	30
Figura 22. Gradiente normal vs. gradiente por lotes. Fuente: Coursera. ....	31
Figura 23. Iteraciones empleando distintos gradientes. Fuente: Coursera. ....	32
Figura 24. Media exponencialmente ponderada. Fuente: Coursera .....	32
Figura 25. Gradiente con impulso. Fuente: Quora .....	33
Figura 26. Decaída de la tasa de aprendizaje. Fuente: ResearchGate .....	34
Figura 27. Clasificador Softmax de tres clases. Fuente: stackoverflow.....	35
Figura 28. Representación por bloques y Softmax. Fuente: Coursera .....	36
Figura 29. Receta para optimización del modelo.....	36
Figura 30. Ejemplo visual de reconocimiento facial en una red convolucional. Fuente: hackernoon.....	37
Figura 31. Ejemplo de representación de imagen en forma matricial. Fuente: Coursera .....	38
Figura 32. Aplicación de filtro. Fuente: Coursera. ....	38
Figura 33. Filtros de convolución. Fuente: UPV .....	39
Figura 34. Aplicación del filtro a una sección de la estructura. Fuente: Coursera.....	39
Figura 35. Red CNN. Fuente: Towards .....	40



Figura 36. Arquitectura LeNet-5. Fuente: researchgate .....	41
Figura 37. Arquitectura AlexNet. Fuente: mdpi .....	41
Figura 38. Arquitectura VGG16. Fuente: Medium .....	41
Figura 39. Bloque residual. Fuente: Wikipedia .....	42
Figura 40. Inception Network. Fuente: Coursera. ....	42
Figura 41. Esquema de la extracción de información.....	45
Figura 42. Esquema del funcionamiento de la aplicación. ....	45
Figura 43: Jerarquía de la base de datos.....	48
Figura 44. Esquema del funcionamiento del Web Scraping. Fuente: edureka! .....	49
Figura 45. Ejecución del Scraper. ....	51
Figura 46. Etapas del reconocimiento facial. Fuente: pyimagesearch.....	52
Figura 47. Modelos de filtros Haar empleados en reconocimiento de objetos. Fuente: unipython .....	53
Figura 48. Aplicación de los bloques sobre un rostro. Fuente: robologs.net .....	54
Figura 49. Resultado de la detección facial en una foto del usuario.....	55
Figura 50. Otro ejemplo de detección facial.....	56
Figura 51. Almacenamiento de las imágenes con las caras resultantes.....	57
Figura 52. Esquema FaceNet. Fuente: arsfutura. ....	59
Figura 53. Generación de embeddings de un usuario. ....	60
Figura 54. Modelos de recortes de caras de distintos usuarios.....	61
Figura 55. Expresión de la distancia Euclídea. ....	62
Figura 56. Expresión del cálculo del peso de los centroides. ....	63
Figura 57. Ejemplo de función con punto de codo. ....	63
Figura 58. Iteraciones de la criba con K-means. ....	65
Figura 59. Transformaciones geométricas. Fuente: arxiv .....	66
Figura 60. Transformaciones fotométricas. Fuente: arxiv.....	66
Figura 61. Ejemplos de transformación de componentes. Fuente: arxiv .....	67
Figura 62. Ejemplos de transformación de atributos. Fuente: arxiv .....	67
Figura 63. Transformaciones mediante imgaug. Fuente: imgaug.....	68
Figura 64. Transformaciones realizadas para el aumento de datos. ....	69
Figura 65. Esquema general de la extracción de información adicional. ....	70
Figura 66. Esquema general de los modelos de red neuronal. Fuente: Packt.....	71
Figura 67. Llamadas recurrentes realizadas al modelo. Fuente: yunjey.....	72
Figura 68. Captura de la disposición del dataset Flickr8K. ....	73
Figura 69. Estructura de los ficheros que contienen las descripciones. ....	73
Figura 70. Estructura de capas del modelo VGG16. Fuente: neurohive.io .....	74
Figura 71. Summay del modelo VGG16.....	75
Figura 72. Ejemplos de aparición. ....	76
Figura 73. Palabras más y menos frecuentes de nuestro vocabulario. ....	77
Figura 74. Visualización de imágenes parecidas entre sí. ....	78

Figura 75. Imágenes próximas.....	79
Figura 76. Estructura del modelo general en detalle.....	80
Figura 77. Variación de la función de coste.....	81
Figura 78. Ejemplo del formato del archivo de descripciones resultante. ....	82
Figura 79. Ejemplo de descripción bien predicha.....	83
Figura 80. Ejemplo de una mala predicción. ....	83
Figura 81. Invocación del objeto CountVectorizer. ....	85
Figura 82. Entrenado del modelo BoW. ....	85
Figura 83. Diccionario de palabras y frecuencias. ....	85
Figura 84. Palabras más repetidas de un usuario. ....	86
Figura 85. Fichero de hobbies. ....	86
Figura 86. Tendencias finales del usuario.....	86
Figura 87. Popularidad. ....	87
Figura 88. Usuarios frecuentes. ....	88
Figura 89. Tags.....	88
Figura 90. Emojis. ....	89
Figura 91. Proceso de comparación de dos rostros mediante FaceNet. Fuente: mc .....	92
Figura 92. Gráfica con los resultados del análisis de eficacia.....	93
Figura 93. Ejemplo de ejecución 1. ....	94
Figura 94. Ejemplo de ejecución 2. ....	95
Figura 95. Ejemplo de ejecución 3. ....	96

# 1-Introducción y motivación

Hoy en día, la sociedad convive en medio de un flujo de datos constante en donde la privacidad juega un papel importante. En pleno siglo XXI, la mayoría de la sociedad tiene un estrecho vínculo con las Redes Sociales, las cuales empleamos como medio para compartir con nuestro entorno el día a día. Pero, ¿realmente somos conscientes del impacto que esto genera para nuestra vida privada?

Muchos datos de los que compartimos dejan de ser completamente nuestros desde el momento que son subidos a la red. Uno de los principales ejes de este proyecto, a parte de la elaboración técnica del mismo, es poner en conocimiento y concienciar del alcance que pueden llegar a tener nuestros datos, ya que con una sola foto y con el debido análisis de datos, se puede llegar a extraer más información de nosotros mismos de la que creemos.

En este caso, me ceñiré a los datos extraídos de la red social Instagram, para posteriormente realizar un reconocimiento facial del usuario mediante redes neuronales convolucionales y una extracción de información de los datos obtenidos como puede ser la obtención de tendencias o aficiones según las fotos del usuario reconocido.

Una vez más, el resultado final de este proyecto, como la mayoría de los procesos de *Deep Learning* que emplean datos personales, puede tener multitud de aplicaciones en el mundo real que a lo largo del proyecto analizaremos, aunque muchas de ellas tengan fines comerciales.

Es por ello por lo que el objetivo principal de este proyecto es crear un sistema de reconocimiento facial, basado en técnicas de *Deep Learning* y a partir de datos obtenidos de Instagram. Para alcanzarlo, hemos dividido este objetivo en dos objetivos particulares:

- Construcción de un sistema de reconocimiento facial, en donde incluimos todas las técnicas de clasificación y visión en reconocimiento de caras a través de redes neuronales.
- Obtención de la base de datos, y todas las técnicas que emplearemos para extracción de información de esta.

Toda la información y conocimiento en torno a las redes neuronales que vamos a emplear ha sido adquirida tras la realización del curso de *Deep Learning.ai* de Coursera, por lo tanto, emplearemos materiales ya trabajados en dicho curso.

## 1.1 Entorno de trabajo

Para la elaboración del proyecto se ha empleado en su totalidad el lenguaje de Python mediante la funcionalidad que ofrece Anaconda con los notebooks de Jupyter, elaborando un notebook diferente para cada etapa del proceso. Todos ellos han sido finalmente unidos para dar como resultado final de este proyecto una aplicación general que realiza todo el proceso paso por paso. En caso de querer analizar con detalle cómo se ha elaborado la solución a cada problema, recurriremos a las implementaciones individuales (a cada notebook) y no al general.

La mayor parte de los modelos empleados durante el desarrollo han sido previamente pre-entrenados o se han descargado los pesos óptimos, pero en algunos casos, como por ejemplo, a la hora de realizar el image captioning, se ha implementado todo el modelo paso a paso haciendo uso de la aplicación de Google Colaboratory para realizar un entrenamiento de la red neuronal y entender mejor el funcionamiento de esta.

Todos los ejemplos de imágenes y la cuenta de Instagram que he empleado para la realización del proyecto inicialmente han sido extraídos de mi cuenta personal, por lo tanto, las imágenes y la información que aportaré a lo largo del proyecto serán sacadas de un individuo real como es mi caso. Una vez concluido el proceso y realizadas las pruebas del correcto funcionamiento de este, iremos añadiendo cuentas de usuarios que accedan a colaborar para realizar una prueba más extensa y generalizada del programa.

## 2-Preliminares

En este apartado del proyecto y para entrar en contexto, explicaremos todos los fundamentos teóricos del principal eje del proceso del reconocimiento facial, las redes neuronales. Al trabajar con imágenes, el proceso se centrará específicamente en el uso de las redes neuronales convolucionales. Ahora veremos el porqué.

### 2.1 Machine Learning

En los últimos años y debido al poder que han adquirido los datos en el denominado efecto Big Data, el Machine Learning ha cobrado gran importancia. El Machine Learning se define como el aprendizaje de las máquinas, incluido dentro del abanico de campos que abarca la Inteligencia Artificial.

Debido a la gran cantidad de datos que se generan día a día y dado la cantidad de información que se puede extraer de ellos, nace la necesidad de tener que procesarlos. Es tanta la información generada, que es prácticamente imposible que los seres humanos podamos tratar manualmente esos datos. De ahí nace la idea del Machine Learning. Máquinas que realicen automáticamente y de forma masiva el trabajo que nosotros no podemos realizar, y que, a partir de multitud de técnicas, estas consigan aprender por sí solas para facilitar las próximas labores de extracción de información.

En términos más precisos, todos los sistemas de Machine Learning se basan en generar un modelo, el cual es el encargado de procesar toda la información y tomar decisiones por sí mismo. Este modelo trabajará obteniendo una entrada de datos, procesándolos, obteniendo una salida sobre estos.

Toda técnica de Inteligencia Artificial tiene en común que intenta emular la inteligencia que aportaría un humano para la realización de esa labor. Lo que diferencia el Machine Learning del resto de técnicas IA es que este realiza un aprendizaje y un actualizado automático de los parámetros de su modelo.

Hoy en día el Machine Learning es una de las técnicas más extendidas del mundo, ya que gracias a todo el trabajo que puede llegar a realizar de forma rápida y precisa, está presente en la mayoría de los servicios, empresas y generalmente en campos donde se trabaje con grandes cantidades de información.

Los problemas o los modelos que aborda el Machine Learning se pueden diferenciar en tres tipos, basándose en la forma en la que estos aprenden:

- Aprendizaje No Supervisado: Los datos con los que el sistema aprende no están previamente etiquetados, es decir, no distinguimos previamente ninguna característica de ellos si no que más bien este tipo de modelos se encarga de procesar los datos, y posteriormente reconocer ciertos patrones dentro del propio conjunto de datos y clasificarlos según lo aprendido. En este proyecto emplearemos una de las técnicas no-supervisadas más extendidas, el *clustering*.
- Aprendizaje Supervisado: A partir de un conjunto de datos previamente etiquetados, el modelo aprende fijándose en ellos. Así, una vez aprendido y ajustados sus parámetros, el modelo será capaz de etiquetar nuevos datos. Dentro del aprendizaje supervisado se diferencian los problemas de clasificación (establecer a qué clase pertenece un objeto, por ejemplo en nuestro caso dadas imágenes clasificar los rostros de la gente) y de regresión (predicción de valores reales, como por ejemplo, el valor de un inmueble). En la mayor parte del proyecto emplearemos este tipo de Machine Learning, ya que las redes neuronales empleadas se basan en estos principios.
- Aprendizaje por Refuerzo: El modelo aprende basándose en anteriores predicciones, es decir, tendrá en cuenta si la predicción ha sido correcta o no para modificar sus parámetros. Se dice que este tipo de aprendizaje está basado en la experiencia, y no necesita una gran cantidad de información como es el caso de los otros dos tipos. Este tipo de técnicas no han sido empleadas en nuestro proyecto.

## 2.2 Deep Learning

Aunque por definición se considere que el Machine Learning engloba campos como el Deep Learning, muchos científicos van más allá y aseguran que es todo lo contrario, ya que, el Deep Learning es más profundo y emplea algoritmos más complejos. El Machine Learning, en cambio, solo se centra en parsear datos y dar una solución a ellos. De cualquier modo, no entraremos en debates, siendo objetivos y explicando en qué se basa el Deep Learning.

El Deep Learning, o Aprendizaje Profundo, trata de emplear técnicas en donde los datos son traspassados a través de numerosas capas (de ahí el nombre de aprendizaje profundo) para realizar el proceso de aprendizaje. Uno de los fundamentos básicos en la creación del Deep Learning fue la analogía con la estructura biológica de las neuronas, y cómo se lleva a cabo el proceso de aprendizaje en los seres humanos. De hecho, muchas

de las cuestiones que trataremos a continuación tienen mucha similitud con la disposición de la inteligencia humana. Una de las aplicaciones más extendidas de hecho, son las redes neuronales.

### 2.2.1 ¿Por qué el Deep Learning está creciendo?

Como ya hemos dicho antes, la cantidad de los datos que nos conciernen día a día es muy grande, y cada año va a más. A medida que pasa el tiempo tenemos más accesibilidad a dispositivos como cámaras, sensores ... Todo ello conlleva al constante crecimiento del número de datos generados.

El Deep Learning, no es algo novedoso, ya que lleva inventado muchos años, solo que cuando disponemos de conjuntos de datos de un tamaño reducido, el rendimiento de algoritmos clásicos de clasificación o regresión viene a ser el mismo que empleando una red neuronal, incluso mejor, ya que a veces la implementación y el coste computacional es menor. Antiguamente y por lo que hemos mencionado anteriormente, la cantidad de datos no era tan grande, lo que nos llevaba a emplear algoritmos sencillos o tradicionales, obteniendo resultados parecidos. Hoy en día, con las cantidades de datos que trabajamos, estos algoritmos se han quedado estancados y no generan un buen rendimiento, lo que nos obliga en la mayoría de los casos a emplear una red neuronal. Cuanto más grandes (o más profundas) sean las redes neuronales, más difíciles serán de implementar y entrenar, pero a su vez, a mayor cantidad de datos, estas ofrecerán un mejor rendimiento.

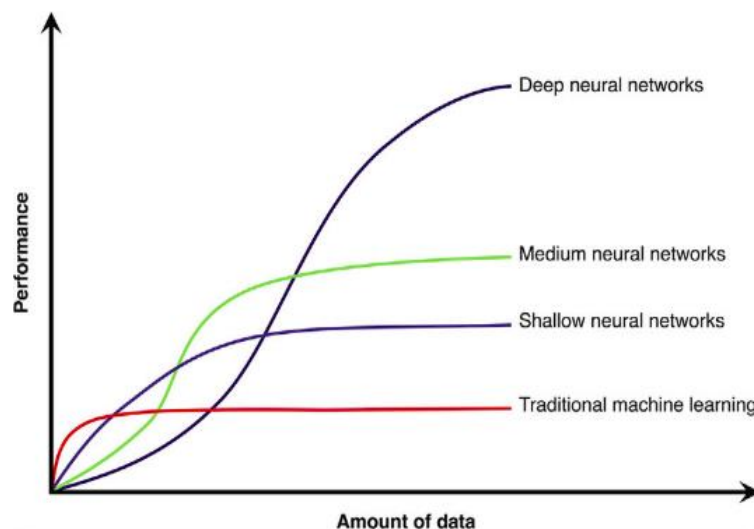


Figura 1. Gráfico del rendimiento de modelos según cantidad de datos empleada. Fuente: [researchgate](https://www.researchgate.net/publication/312211111)

Debido al rendimiento que ofrecen las redes neuronales, y debido también a que las empresas cada vez necesitan procesar de manera más eficiente sus datos, se puede decir que las redes neuronales están en pleno auge comercial (lo único necesario para realizar una red neuronal eficiente es un conjunto grande de datos previamente etiquetados, y una red neuronal extensa).

### 2.2.2 Proceso iterativo

Toda red neuronal requiere de unos recursos y un proceso, para que antes de usarla, esta pueda dar buenos resultados y ser productiva. Pero la rapidez en la realización de este proceso iterativo dependerá de varios factores:

- Datos: Necesitamos disponer de una fuente de datos (o comúnmente llamados *datasets*), y que estos hayan sido previamente etiquetados con lo que queremos que nuestro modelo aprenda. Quizás el proceso de etiquetado sea uno de los más sencillos, pero a su vez más costosos de todo el proceso general, ya que este requiere que alguien lo haya correctamente clasificado previamente, y dado el gran tamaño de los conjuntos de datos, esto conlleva muchísimo tiempo. Existen técnicas para ahorrar mano de obra humana en este proceso de etiquetado, pero todavía no se ha conseguido que esto se pueda hacer de forma automática.
- Computación: El proceso de computación es aquel que es realizado por parte de la máquina, es decir todo el tiempo en el que la máquina procesa los datos y actualiza valores o realiza cálculos. Este factor dependerá de lo material, es decir, si los recursos empleados son buenos, esto hará que este proceso se realice de forma más efectiva.
- Algoritmos: Los algoritmos internos que empleemos para el cálculo de valores, el usar unos u otros según los tipos de datos o resultados en general también afectarán al proceso, ya que todos ellos no tienen la misma complejidad de ejecución.

Una vez dicho esto, podemos reflejar el proceso de aprendizaje iterativo, en forma de rueda (Figura 2), ya que es un proceso que nunca acaba. Es decir, una red neuronal no es igual de eficiente siempre, al igual que los datos cambian con el paso de los años. Es por eso por lo que esta se reentrenará y se adaptará gracias a este proceso iterativo.

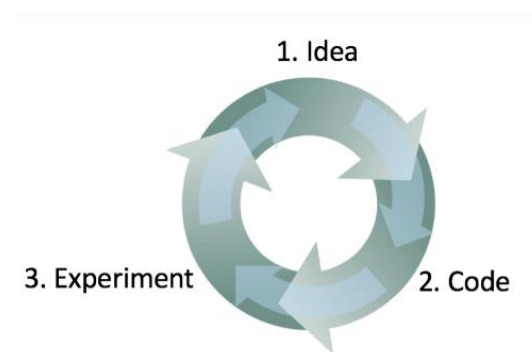


Figura 2. Proceso de aprendizaje iterativo. Fuente: [medium](https://medium.com)



### 2.2.3 Redes Neuronales

Una vez que conocemos el alcance y el modo en el que el *Machine Learning* resuelve sus problemas, vamos a ser más precisos y vamos a centrarnos en entender el concepto de las redes neuronales.

#### Introducción

Las redes neuronales no son más que un tipo de modelo empleado para la resolución de problemas de *Machine Learning*. El objetivo de estas, como todo algoritmo de aprendizaje, es dar una predicción sobre una entrada concreta basándose en unos datos previamente aprendidos. Estos datos son procesados a través de neuronas, que interconectadas entre sí forman la llamada red neuronal. Mientras más neuronas haya conectadas, más profunda será la red formada. Las neuronas, simplemente son una analogía humana, que lo que intentan es modelar el comportamiento que tendría una neurona natural y son una unidad esencial de las redes neuronales.

- Entrada: Se trata de la información que usaremos para predecir valores. Usualmente estos datos pertenecen al conjunto  $X$ , y a cada dato se le asigna el nombre de  $X^{(i)}$ , y se le conocerá como el ejemplo  $i$ -ésimo. Cada objeto o ejemplo se distinguirá por sus características (las cuáles serán las que pasemos a la red), y se denotará  $N$  como el número total de características ( $x_1 \dots x_n$ ). Por lo tanto, podemos decir que  $x_j^{(i)}$  hace referencia a la característica  $j$  del ejemplo  $i$ -ésimo. El conjunto de entrenamiento estará compuesto por  $M$  ejemplos, y todos ellos se almacenarán en una matriz de dimensión  $(N \times M)$ .
- Salida: Es el resultado que devolverá la red una vez haya procesado la información. La salida se denotará como  $Y$ , y sus dimensiones dependerán de la estructura de la red que hayamos empleado.
- Pesos: Son los valores que adquieren los pesos de cada neurona. Estos se van actualizando en cada iteración y su función es almacenar los valores que actuarán en la aplicación lineal de los datos. Es decir, la salida dependerá del valor de los pesos. Estos hacen referencia al enlace entre una característica de entrada y la propia neurona, siendo  $w_i$  el peso entre la  $i$ -ésima característica y la neurona.
- Neurona: También conocido como perceptrón, es el elemento más pequeño que se encarga de procesar una información. Esta recibe información de entrada (variables de entrada o características), procesa la información (aplicando los pesos) y finalmente da una salida o predicción. Engloba todos los conceptos que acabamos de explicar. Esta lo que hace principalmente es realizar la suma ponderada de las características y los pesos, aplica una función de activación (las veremos más adelante) y devuelve una salida.

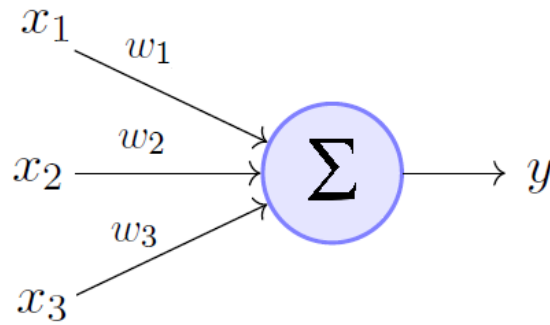


Figura 3. Representación visual de un perceptrón. Fuente: [Alberto Torres](#)

En la representación gráfica de la Figura 3, podemos apreciar las características de entrada ( $x_1$ ,  $x_2$ ,  $x_3$ ), los pesos aplicados ( $w_1$ ,  $w_2$ ,  $w_3$ ) y la variable de salida ( $y$ ). A esta variable de salida, si va a ser empleada como la entrada a una nueva neurona enlazada, la denominaremos activación. Para el cálculo de la salida, aplicaremos la función:

$$y = a(z)$$

$$z = w_0x_0 + \dots + w_nx_n + b$$

Estas ecuaciones realizan la predicción de salida de una neurona. Si nos fijamos bien, esto se parece mucho a la ecuación utilizada en regresión logística. Con el uso de una sola neurona como esta solo podemos hacer frente a problemas linealmente separables. Para poder abordar problemas más complejos, necesitamos enlazar más neuronas como esta, creando así las redes multicapa. En la parte de Redes Profundas analizaremos esto.

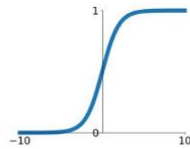
La función  $a$  hace referencia a una función de activación. Estas son empleadas para el cálculo final de la predicción. En clasificación binaria, por ejemplo, aplicando esta función conseguimos obtener la predicción en el intervalo determinado de valores comprendidos entre cero y uno.

### Funciones de activación

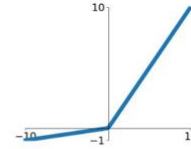
Existen muchas funciones de activación, muchas de ellas no lineales, y no existe ningún protocolo para determinar cuál funciona mejor. La elección de una u otra esencialmente dependerá del rango en el que se encuentren los datos de ejemplo. A continuación, mostramos algunos ejemplos (Figura 4).

**Sigmoid**

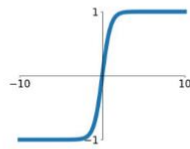
$$\sigma(x) = \frac{1}{1+e^{-x}}$$

**Leaky ReLU**

$$\max(0.1x, x)$$

**tanh**

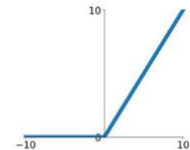
$$\tanh(x)$$

**Maxout**

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

**ReLU**

$$\max(0, x)$$

**ELU**

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$

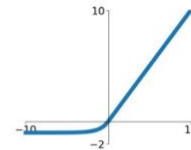


Figura 4. Funciones de activación más comunes. Fuente: [medium](https://medium.com/@dimitrybaranovskiy/activation-functions-101-400000000000)

La función de activación más empleada y que mejor funciona es la *ReLU*, aunque en cada capa de la red neuronal tenemos libertad de elegir cuál usar. De hecho, es recomendable emplear la función Sigmoide para la última capa de salida (siempre que la clasificación sea binaria, es decir, haya dos clases de salida). En caso de que la salida sea multiclase, como veremos más adelante, emplearemos *softmax*.

### Descenso por gradiente y actualización de pesos

Uno de los objetivos más importantes de las redes neuronales es buscar un buen ajuste de los pesos, de tal manera que generalice el problema de la mejor forma posible, consiguiendo que se produzca la menor desviación a la hora de calcular nuevas predicciones, o, dicho de otra forma, que el modelo sea lo más efectivo y eficaz posible.

El error producido a la hora de hacer predicciones se mide a través de una función de coste, o en inglés, *loss function* (en el caso de la regresión). Al igual que las funciones de activación, se pueden emplear distintas funciones de coste. La más popular y la que nosotros emplearemos será el error cuadrático medio (MSE). Esta viene dada por la expresión:

$$MSE = \frac{1}{M} \sum_{i=1}^M (real_i - estimado_i)^2$$

En donde M es el número de ejemplos del conjunto de datos, *real* es el valor de salida que da ese ejemplo, y *estimado* la predicción que realiza el modelo ante ese ejemplo.

En el caso de la clasificación, empleamos para calcular el coste una función llamada *cross-entropy*:

$$CCE(p, t) = - \sum_{c=1}^C t_{o,c} \log(p_{o,c})$$

Obviamente, el error es bastante intuitivo, ya que simplemente se trata de la suma de las veces que el modelo ha predicho mal la salida en todo el conjunto de datos. ¿Cómo reducimos este error? Lo único que debemos hacer es ir actualizando los pesos de forma iterativa, de modo que en cada iteración el error vaya disminuyendo. Esto, como la mayoría de los algoritmos que buscan la solución óptima de un problema, se basa en encontrar la solución por medio del gradiente. A continuación, explicaremos resumidamente el algoritmo que tantas veces hemos trabajado a lo largo de la carrera.

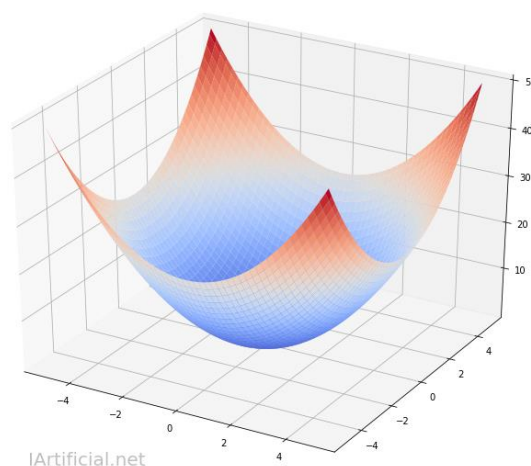


Figura 5. Representación gráfica de la función de coste. Fuente: [IArtificial](http://IArtificial.net)

El gradiente, dada una función, tiene como objetivo encontrar el valor mínimo de esa función de forma iterativa. La función, en este caso, es la función de error que hemos mencionado antes. Es decir, en cada iteración del algoritmo, este tratará de actualizar los valores de los pesos de tal manera que cada vez nos acerquemos más y más al mínimo de la función.

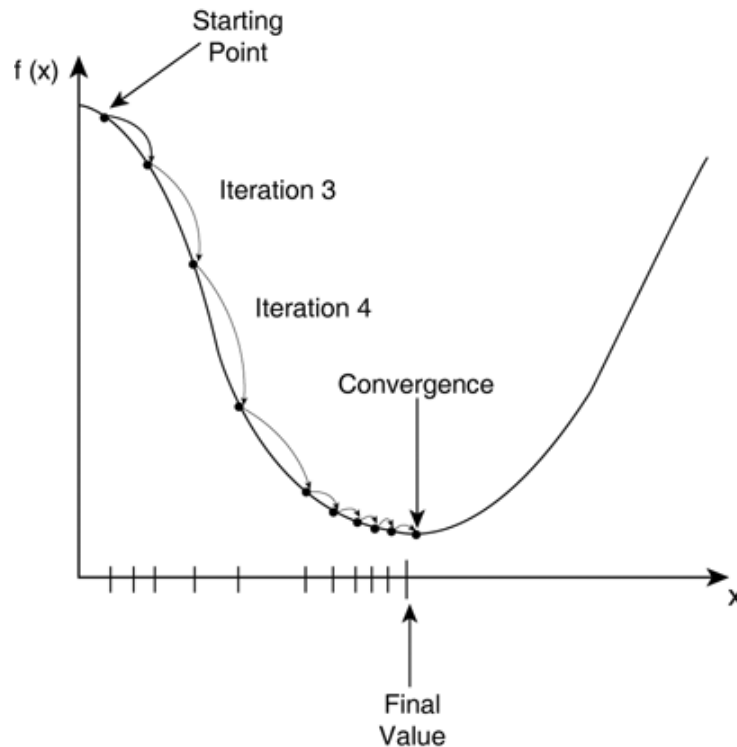


Figura 6. Iteraciones del gradiente. Fuente: [cs.eu](http://cs.eu)

En la Figura 6, mostramos las iteraciones realizadas en plano hasta llegar al mínimo de la función. Cada salto representa una iteración del algoritmo. La longitud de estos saltos, o, mejor dicho, la velocidad de convergencia viene dado por el parámetro  $\alpha$ . Este marcará el ritmo de convergencia a la hora de actualizar los pesos. Necesitamos encontrar un ajuste de este parámetro óptimo, para evitar que el algoritmo realice saltos muy pequeños (ralentización del algoritmo), o en su contra, de saltos muy grandes, lo que puede ocasionar que no encontremos nunca el mínimo.

Para ello, recurrimos al uso de las derivadas para actualizar dichos pesos. Como ya sabemos, la función de coste convergerá siempre en el único mínimo que tiene la función, evitando así posibles bucles o entrar en mínimos locales. Las derivadas nos marcarán la dirección de máxima pendiente, o lo que es lo mismo, la dirección en la que queremos descender si queremos encontrar el mínimo. Estas derivadas se realizan del peso a actualizar sobre la función de coste  $J$ :

$$w_i^{new} = w_i^{old} - \alpha \frac{\partial J}{\partial w_i}$$

$$b_i^{new} = b_i^{old} - \alpha \frac{\partial J}{\partial b_i}$$

En el caso de las redes neuronales, emplearemos el descenso por gradiente para el entrenamiento de la red, y para el ajuste óptimo de los pesos.

#### 2.2.4 Redes Neuronales Profundas

Hasta ahora hemos visto cómo cada neurona puede ajustar sus parámetros a partir de un conjunto de entrada, para después dar una predicción mediante las funciones de activación. También hemos visto que las neuronas pueden agruparse, siendo las activaciones de unas, las entradas para otras nuevas. A continuación, introduciremos el término de capa o *layer* para explicar mejor el término de red profunda.

##### Introducción

Una red neuronal profunda es una estructura, que es el resultado de enlazar una cantidad relevante de neuronas entre sí, dando posibles soluciones a problemas más complejos y no-lineales.

Se introduce el término de capa (número total de capas del modelo,  $L$ ), para hacer referencia a los distintos niveles que se forman de activaciones una vez hayamos enlazado unas neuronas con otras. Si nos fijamos en la siguiente imagen, en donde se muestra un ejemplo visual de una red neuronal profunda, es más sencillo distinguir las distintas capas que forman el sistema.

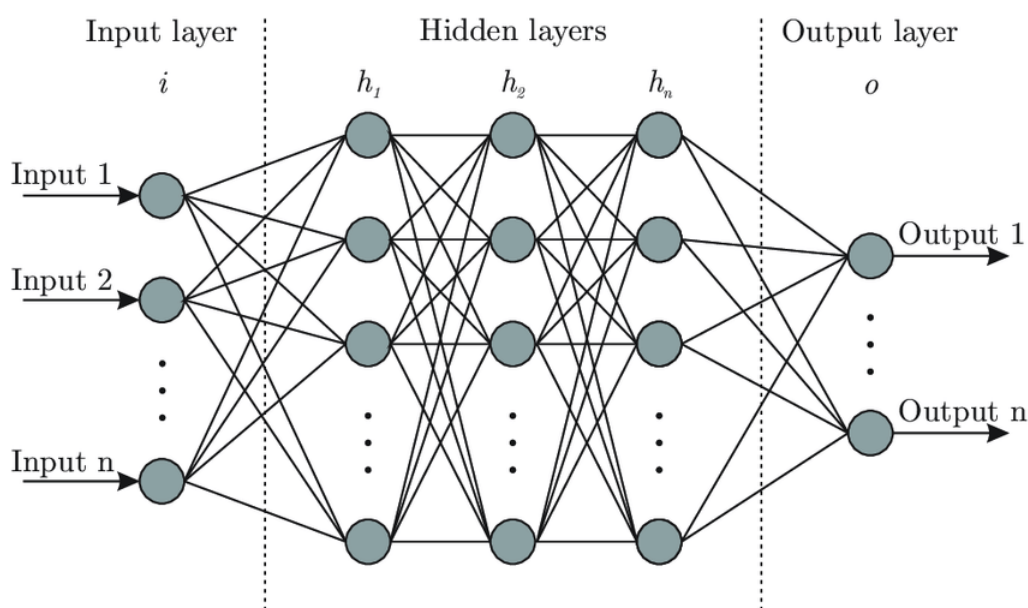


Figura 7. Esquema de una red multicapa. Fuente: [ResearchGate](https://www.researchgate.net/publication/312111111)

Si nos fijamos en el esquema de la Figura 7, el sistema multicapa de una red neuronal se compondrá de una capa de salida (por donde se introducirá la información al sistema), capas ocultas (en este caso tenemos tres capas ocultas, pero mientras más capas ocultas más profunda se dirá que es la red) y la capa de salida (por donde obtendremos las predicciones finales). A partir de ahora, las activaciones y los pesos de cada capa serán almacenados de forma matricial, para que el cálculo sea más eficiente.

La forma en la que calculamos las activaciones de cada neurona es también intuitiva, ya que tenemos que realizar el mismo proceso que hacíamos con una sola neurona, solo que ahora forman parte del cálculo los pesos de los enlaces de otras neuronas también, reduciéndose en la expresión (recordando que ahora emplearemos matrices):

$$\mathbf{A}^{[i]} = \mathbf{a}(\mathbf{Z}^{[i]})$$

$$\mathbf{Z}^{[i]} = \mathbf{W}^{[i]} \mathbf{X}^{[i]} + \mathbf{b}^{[i]} \quad \text{en donde} \quad \mathbf{X}^{[i]} = \mathbf{A}^{[i-1]}$$

Obviamente, las entradas a una neurona equivalen al resultado de la activación de la capa anterior. El gradiente para la actualización de pesos se aplica de la misma forma solo que esta vez recorreremos toda la red actualizando los pesos.

### Forward y backward propagation

Una vez vayamos a implementar el gradiente, si tuviéramos que calcular las derivadas de todos los pesos en cada iteración, esto sería muy ineficiente. Por ello, a la vez que calculamos las activaciones en el proceso de propagación hacia delante (*forward propagation*) en la red, almacenamos el valor de estas para emplearlas posteriormente en el proceso contrario, de propagación hacia atrás (*backpropagation*) para la actualización de los pesos. Mediante la representación de la red a través de bloques, este proceso se hace más intuitivo:

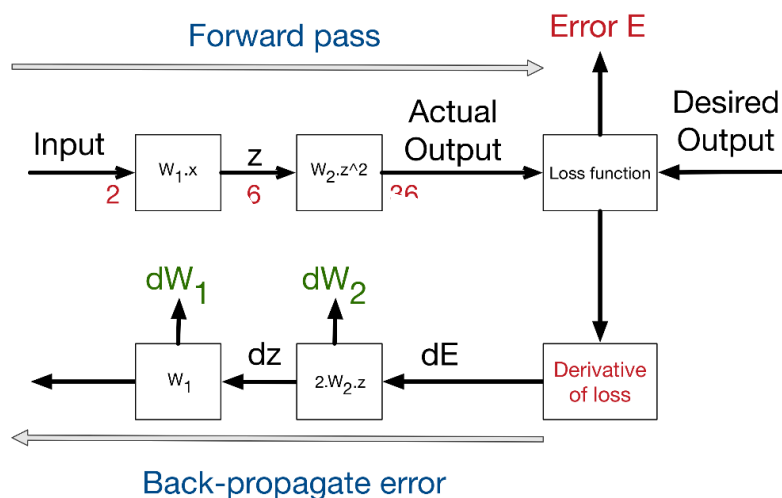


Figura 8. Representación por bloques. Fuente: [Medium](#)



### Parámetros – Hiperparámetros

Los parámetros son todas aquellas variables que forman parte en el proceso del cómputo de la predicción del modelo, y las cuales no pueden ser definidas por el usuario (como, por ejemplo, los pesos).

Los hiperparámetros son todos aquellos parámetros cuyo valor queda en manos del usuario, los cuales necesitamos definir de forma ajena al modelo, pero que de ellos también dependerá el rendimiento del sistema (tasa de aprendizaje  $\alpha$ , número de iteraciones del gradiente, número de capas del modelo  $L$ , elección de la función de activación  $a...$ ).

#### 2.2.5 Optimización del modelo

Llegados a este punto ya conocemos la forma en la que funciona una red neuronal profunda, pero eso no implica que esta funcione de manera óptima. El objetivo de este apartado es tener una idea general de en qué se basa la optimización en redes neuronales, y conocer las principales técnicas que se emplean para ello, sin entrar en mucho nivel de detalle.

El objetivo, como en todos los sistemas de clasificación y aprendizaje, es evaluar el modelo mediante los conjuntos de *train*, *test* y validación, para así intentar obtener el mejor rendimiento posible.

### Bias – Varianza

Se trata de dos términos importantes cuando hablamos de capacidad de generalización en un algoritmo de aprendizaje. La varianza, mide el nivel de ajuste que tiene el modelo entrenado sobre los datos de entrenamiento. Mientras que el *bias*, o el sesgo, trata de medir la dispersión de estos o lo poco que se ajustan al modelo. En la Figura 9 obtenemos una representación gráfica de cómo afectan los términos al modelo.

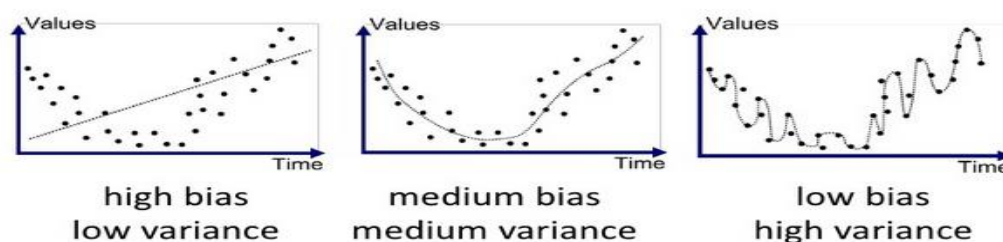


Figura 9. Bias y varianza. Fuente: [lokad](#)



Los humanos, en un clasificador común como puede ser el reconocimiento de gatos en fotos, obtenemos un error de en torno al 0%. Según Bayes, este error óptimo es casi imposible de encontrar, pero intentaremos buscar uno parecido buscando el equilibrio de *bias* y *varianza*.

### Regularización

Cuando nos encontramos ante un modelo con un grado elevado de varianza, esto querrá decir que el modelo que hemos generado se ajusta demasiado a los datos de entrenamiento. A esto lo definiremos como sobre-entrenamiento o sobreaprendizaje (en inglés, *overfitting*). Ante los datos que hemos entrenado el modelo, se ajusta perfectamente a ellos, produciéndose un error muy cercano a cero. Esto se puede convertir en un arma de doble filo, ya que el modelo, pierde capacidad de generalización pudiendo predecir mal futuros ejemplos.

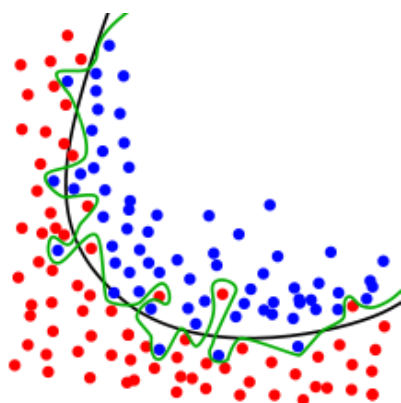


Figura 10. Modelo sobre-entrenado.  
Fuente: [Wikipedia](#)

En la Figura 10 de la izquierda, en verde podemos observar la frontera de decisión (función que separa la elección de clase en un clasificador), la cual se ajusta totalmente a los datos de entrenamiento acertando todos ellos. En futuros datos a predecir, la capacidad de generalización no será buena, pudiendo generar conflictos y menor precisión de acierto. La función en negro, a pesar de realizar más fallos en el conjunto de entrenamiento, abordará mejor futuras predicciones.

Para evitar que aparezca este sobre-aprendizaje en nuestros modelos, una de las soluciones más aplicadas es la regularización. Esta se basa en añadir un pequeño desvío a los pesos en el proceso de aprendizaje, de tal manera que estos no se ajusten del todo a los datos.

La forma de aplicar la técnica es modificar la función de coste del modelo sumando una expresión que dependerá del parámetro de regularización lambda ( $\lambda$ ):

$$L(x, y) \equiv \sum_{i=1}^n (y_i - h_{\theta}(x_i))^2 + \lambda \sum_{i=1}^n \theta_i^2$$

Figura 11. Función de coste con regularización. Fuente: Coursera

El cuadro remarcado hace referencia a la expresión añadida a la función de coste. A esta expresión se la define como regularización L2. También existe la regularización L1, la

cual se usa cuando el número total de pesos de la red es pequeño, pero en la práctica, al no darse el caso, casi no se utiliza:

$$J(\theta) = \frac{1}{m} \sum_{i=1}^m \text{Cost}(h_{\theta}(x^{(i)}), y^{(i)}) + \frac{\lambda}{m} \sum_{j=1}^n |\theta_j|$$

Figura 12. Regularización L1. Fuente: [Coursera](#).

Una vez se haya modificado la función de coste, démonos cuenta que la actualización de pesos del gradiente también debe ser modificada, ya que las derivadas parciales que dependen de los pesos se ven repercutidos por esta nueva expresión.

$$\theta_j := \theta_j - \alpha \left[ \left( \frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)}) x_j^{(i)} \right) + \frac{\lambda}{m} \theta_j \right]$$

Figura 13. Actualización de pesos con regularización. Fuente: [Coursera](#).

**Importancia del valor de  $\lambda$ :** Si nos fijamos en la expresión, la función que cumple el valor de lambda es claro. Trata de aplicar un desvío, de tal manera que cuanto más grande sea ese valor del desvío, menos importancia tendrán los pesos. No hemos nombrado nada de los pesos Bías, ya que estos, normalmente no se regularizan. Si el valor de  $\lambda$  es muy grande, los pesos desaparecen dando lugar a una regresión logística. Pero debemos tener cuidado, ya que, si el valor del parámetro de regularización es elevado, el modelo pasaría de estar sobre-entrenado, a no ajustarse a los datos, por ello, la solución es encontrar un valor de lambda intermedio.

Para comprender mejor por qué los valores altos de regularización conllevan un mal ajuste a los datos, lo explicaremos mediante la función de tangente hiperbólica ( $\tanh$ ):

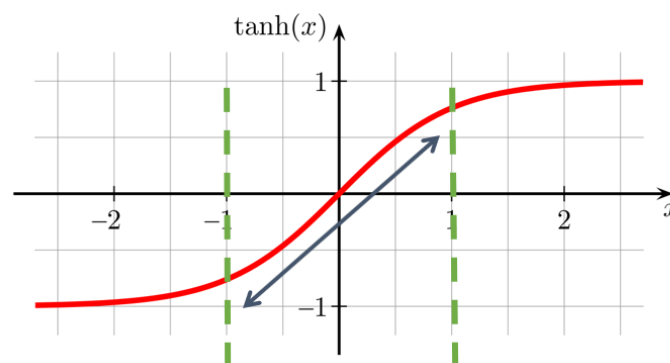


Figura 14. Función de tangente hiperbólica y regularización.

Siendo  $x$  el resultado a la función de activación empleada, analizaremos sus valores. Si limitamos la hipótesis a un único peso para comprender mejor el funcionamiento,

$x = W^*a + b$ . Como hemos explicado antes, los valores altos de  $\lambda$  provocan que los pesos tengan menor importancia incluso llegando a desaparecer, haciendo que  $x$  adopte valores muy bajos. Si limitamos la función  $\tanh(x)$  a valores de  $x$  pequeños (área marcada en la función), obtenemos una función lineal y no se ajustará a valores complejos, teniendo un modelo que no se ajuste a los datos.

**Tipos de regularización:** A continuación, y de forma general nombraremos las distintas técnicas que existen para aplicar la regularización más allá de la implementación teórica de la misma:

- *Dropout regularization* o regularización por abandono: El objetivo de esta técnica consiste en el “abandono” de uno o varios nodos de una capa de la red neuronal a la hora de ser entrenada con los ejemplos de entrenamiento. Dicho de otra forma, por cada ejemplo de entrenamiento, cada capa tendrá una probabilidad de no tener en cuenta los pesos de algún nodo (siendo  $\lambda$  muy grande para estos nodos). De esta forma evitamos el sobreajuste creando pequeñas subredes de entrenamiento dentro de un solo modelo. En el entorno de visión artificial, esta técnica de regularización es la más empleada, ya que a la hora de entrenar modelos nos permite no tener en cuenta todos los píxeles de las imágenes y evitar que los clasificadores sean sobre-entrenados. A la hora de implementar este tipo de regularización, existe una variable llamada *keep.probs*, en donde almacenaremos en un *array* las probabilidades que tiene cada capa de sufrir un abandono de alguno de sus nodos.

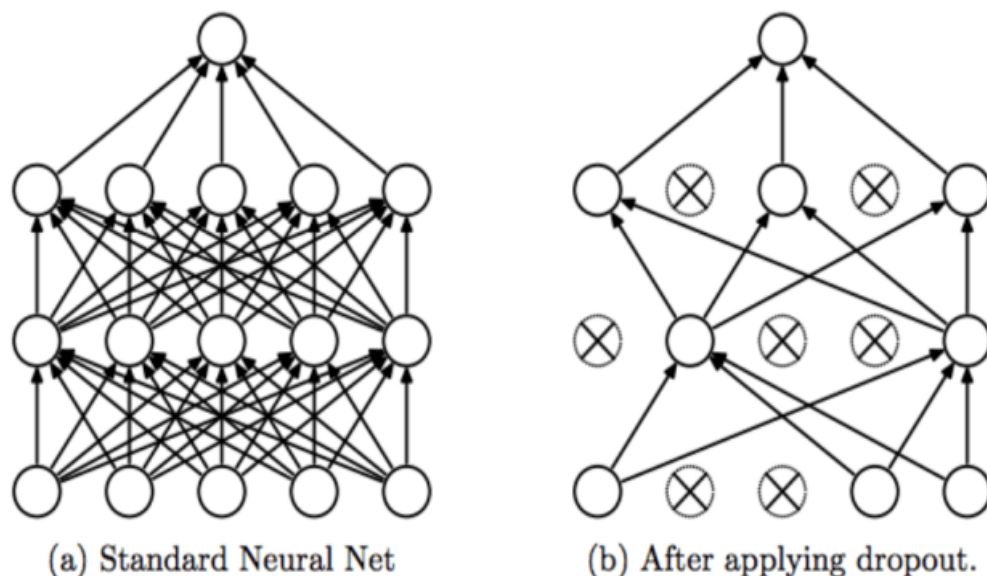


Figura 15. Regularización por abandono. Fuente: [Tech-Quantum](#)

- *Data augmentation*: Esta técnica es muy intuitiva. Consiste en aumentar el tamaño de los datos de entrenamiento (más adelante analizaremos esto con más profundidad), para evitar el sobreajuste.
- *Early stopping* o regularización por parada temprana: Consiste en interrumpir el entrenamiento de los datos en una iteración concreta para evitar que los valores de los pesos se sigan ajustando más de lo necesario. Basándonos en el concepto

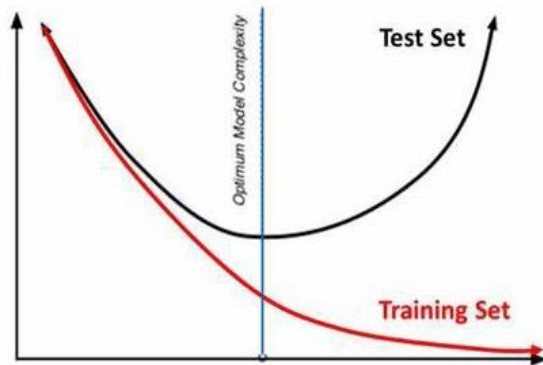


Figura 16. Regularización por parada temprana. Fuente: [Medium](#)

de generalización, nos interesa encontrar el punto en el que los errores en el conjunto de entrenamiento y de *test* son parecidos, pero empiezan a disiparse. El principal inconveniente de esto es la ortogonalización (empleando esta técnica no podemos encontrar el coste mínimo nunca).

### Normalización

La normalización es un concepto que se ha tratado a lo largo de la carrera, bien en teorías de conjuntos o en conceptos de aprendizaje automático. Esta consiste en adaptar los datos numéricos de tal forma que todos ellos pertenezcan a un mismo rango de valores.

Pero, ¿por qué debemos normalizar los datos? Las características de los ejemplos muchas veces pertenecen a rangos de valores muy distintos entre sí, por lo que a la hora de entrenar los pesos algunos podrían tener valores muy elevados referentes a los valores de rango alto, y otros sin embargo valores muy pequeños. Esto si lo trasladamos al fundamento teórico, provoca que obtengamos una función de coste con forma de tazón más alargado (ver comparativa más abajo, Figura 17), lo que hace más costosa la ejecución del descenso por gradiente, teniendo más iteraciones, mientras que con los datos normalizados realizar el descenso resulta más sencillo, ya que los pasos a realizar son uniformes y la búsqueda es más directa.

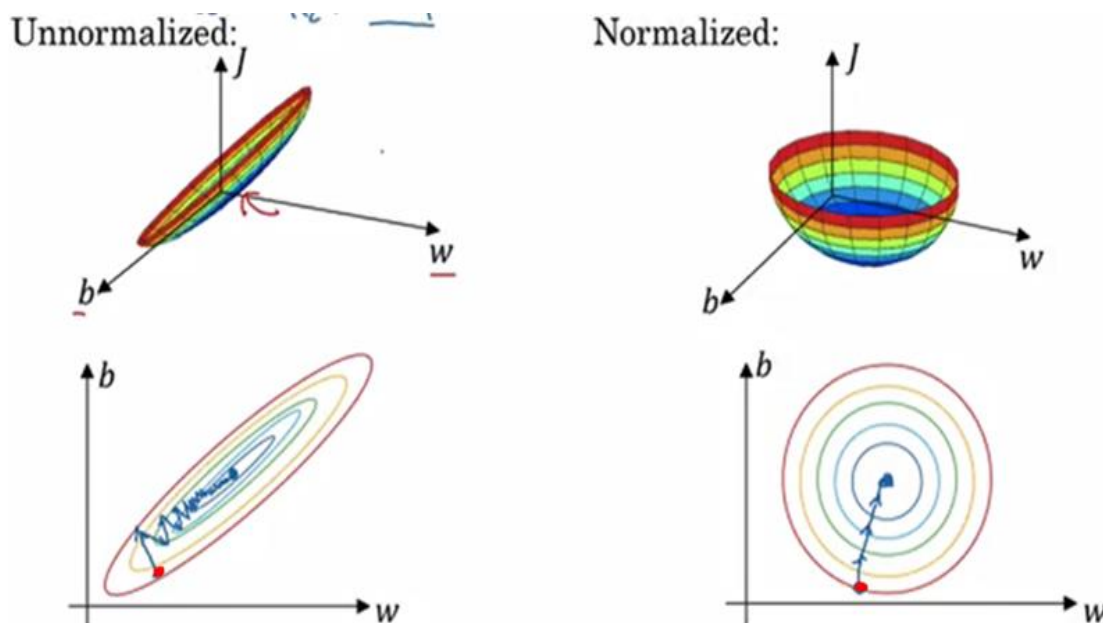


Figura 17. Coste no-normalizado vs. normalizado. Fuente: [Coursera](#).

Como podemos observar en las representaciones en plano, necesitamos muchas más iteraciones para encontrar el mínimo de la función, lo que implica un valor más alto de  $\alpha$  para poder reducir el número total de iteraciones, corriendo el riesgo de que la función no minimice correctamente.

Para llevar a cabo la normalización de los datos de entrada, esta se realiza en dos pasos:

- 1- Restar la media del conjunto: calculamos y restamos la media a cada ejemplo del conjunto de entrenamiento.

$$\bar{X} = \frac{\sum_{i=1}^n x_i}{n}$$

Figura 18. Media aritmética del conjunto de entrenamiento. Fuente: [Coursera](#).

- 2- Normalizar la varianza para agrupar los valores a un mismo rango y evitar la dispersión de los datos.

$$\sigma = \sqrt{\frac{\sum_i^N (X_i - \bar{X})^2}{N}}$$

Figura 19. Desviación típica del conjunto de datos. Fuente: [Coursera](#).

Si representemos estos dos pasos anteriores de forma visual sobre un conjunto de datos de ejemplo:

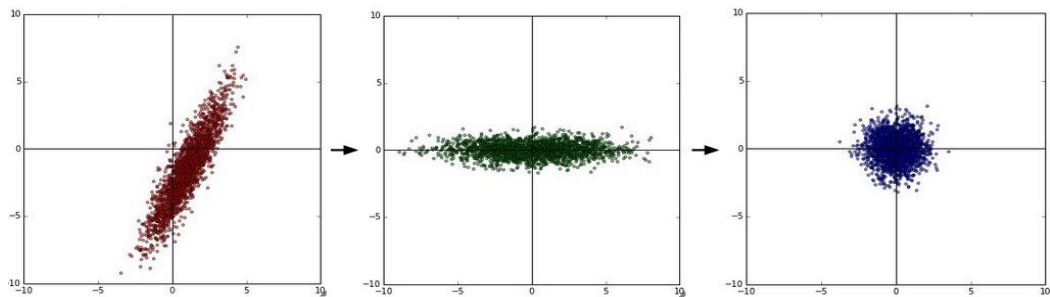


Figura 20. Pasos de la Normalización. Fuente: [Coursera](#)

Y si unimos estos pasos, conseguimos una expresión general para el cálculo de la normalización de todo el conjunto de datos X:

$$z_i = \frac{x_i - \bar{x}}{\sigma}$$

Figura 21. Expresión del proceso de Normalización. Fuente: [Coursera](#).

### Algoritmos de optimización

Cuando se trabaja con una gran cantidad de datos como es el caso de las redes neuronales profundas, es importante emplear algoritmos de optimización para conseguir mejorar el rendimiento del modelo, y en general, del equipo. Para ello e igual que hicimos con las técnicas de regularización, haremos un repaso de los algoritmos de optimización más empleados en *Deep Learning*.

- Descenso por gradiente por lotes (*Mini-batch gradient descent*): Esta técnica de optimización es empleada para reducir el coste computacional del algoritmo de descenso por gradiente original (o *Batch gradient descent*) y su principal objetivo es el empleo de la vectorización para el ahorro de tiempo de ejecución.

Imaginemos que necesitamos entrenar una red neuronal de unos cincuenta millones de datos. En cada ejecución del gradiente normal necesitamos hacer un recorrido por todo el conjunto de datos, lo que ralentizaría el entrenamiento. El gradiente por lotes ofrece una solución a este problema de ralentizado. El objetivo de este es partir el conjunto completo de datos en subconjuntos (*mini-batches*), de forma que en cada iteración se procesa cada subconjunto y se actualizan los pesos, en lugar de tener que usar en cada iteración todos los ejemplos de entrenamiento.

Si ilustramos la función de coste respecto al número de subconjuntos, vemos que, en el caso del gradiente por *mini batch*, se obtiene un mayor ruido en los resultados referentes al coste, pero igualmente desciende hacia una solución de forma más eficiente.

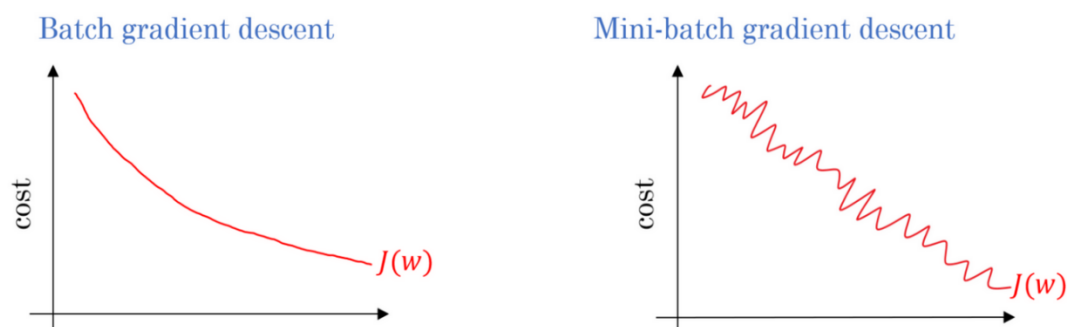


Figura 22. Gradiente normal vs. gradiente por lotes. Fuente: [Coursera](#).

A la hora de emplear esta técnica, la eficiencia dependerá en parte del valor que asignemos al número de *mini-batches* en los que queramos partir el conjunto original de datos. Si este valor es muy cercano al tamaño del conjunto de datos, es decir, solo existe un solo lote, estaremos aplicando el gradiente original. En cambio, si cada lote contiene exactamente un ejemplo, tendremos tantos *mini-batches* como ejemplos haya en el conjunto, lo que da lugar al denominado gradiente estocástico, siendo este más aleatorio y pudiendo ser a su vez más



rápido, pero teniendo el peligro de no converger. La clave es encontrar un número de lotes intermedio para emplear esta técnica.

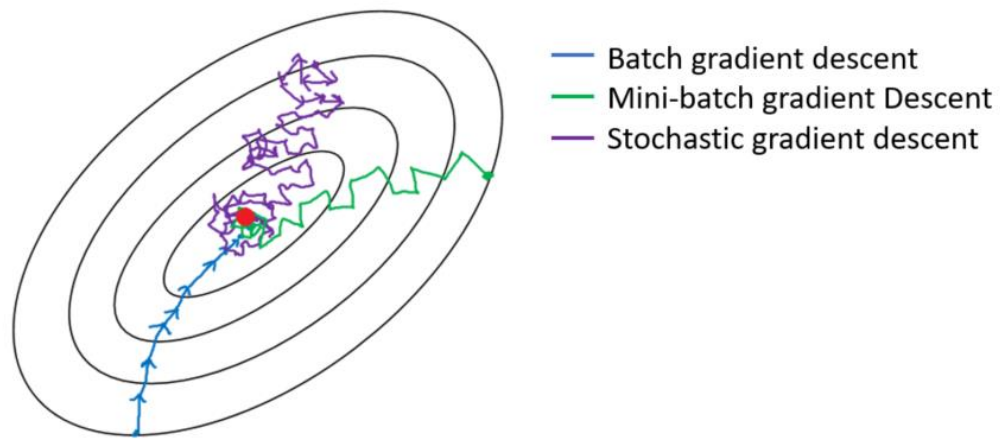


Figura 23. Iteraciones empleando distintos gradientes. Fuente: [Coursera](#).

Como norma general y para mantener la máxima eficiencia con el uso de cada gradiente, se empleará el descenso por gradiente original cuando el tamaño del conjunto de datos no supere los dos mil ejemplos, y el descenso por gradiente por lotes, cuando el conjunto de datos supere esa cifra. El número de *mini-batches* siempre será potencia de dos, y respectivamente se habrá comprobado que esto encaja con la CPU o GPU que realizará el cálculo. A pesar de la eficacia de este, entre los algoritmos de optimización no se trata de los más efectivos.

- Promedios exponencialmente ponderados (*Exponentially Weighted Averages*): Esta técnica se emplea cuando los datos del modelo presentan una tendencia específica. Por ejemplo, si representamos las temperaturas del año, lógicamente los pesos referentes a verano adoptarán unos valores más altos. De esta forma, el error producido es menor con respecto al empleo de la media estándar.

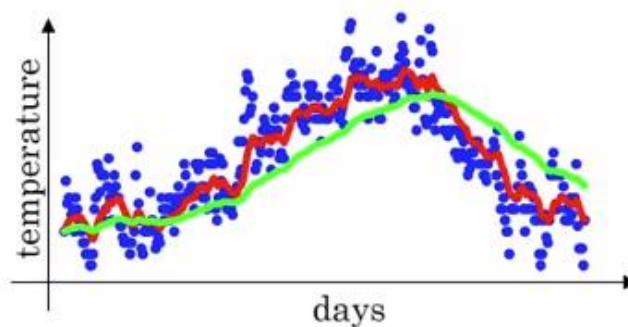


Figura 24. Media exponencialmente ponderada. Fuente: [Coursera](#)



- **Descenso por gradiente con impulso (*Momentum*):** Este algoritmo de optimización consiste en calcular un promedio ponderado de sus gradientes y emplear este para actualizar los pesos de una forma más generalizada. Lo que este nos proporciona es tomar un camino horizontalmente más directo para llegar hasta el mínimo. Mediante este algoritmo, optimizamos el total de iteraciones que se van a realizar posteriormente para entrenar el modelo, pero para encontrar el camino, previamente necesitamos realizar un coste extra calculando el promedio ponderado.

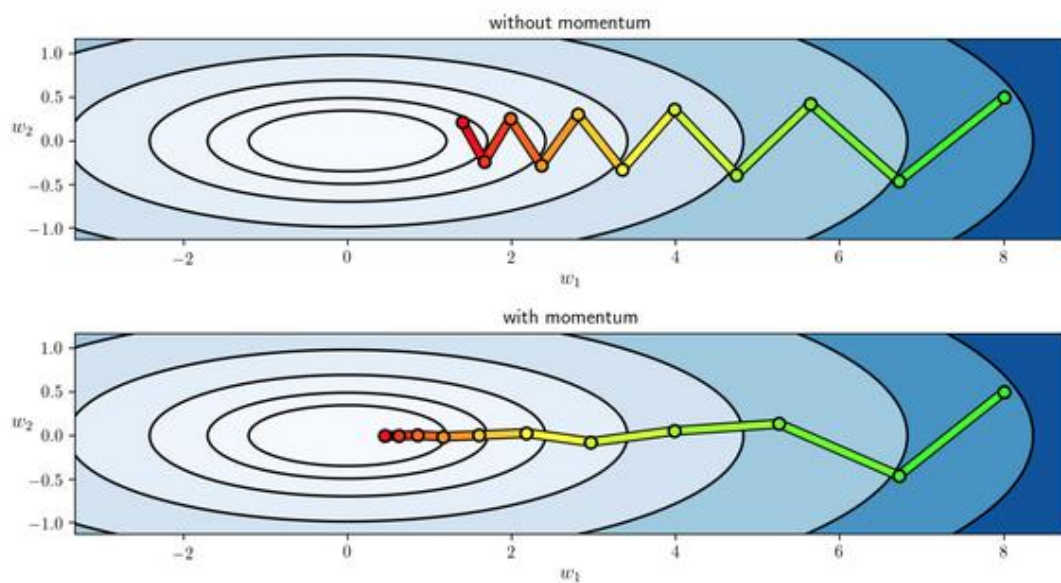


Figura 25. Gradiente con impulso. Fuente: [Quora](#)

El también conocido algoritmo RMSprop, tiene el mismo objetivo que el cálculo del gradiente con impulso, solo en este, el proceso es distinto ya que, en lugar de calcular las medias ponderadas, trabaja con las derivadas parciales de cada peso.

- **Algoritmo Adam:** Este algoritmo trata de combinar los últimos dos mencionados, el gradiente con impulso y el RMSprop, para así acelerar todavía más la propagación horizontal y minimizar el número de iteraciones. Lo que el algoritmo realiza en cada iteración es dar una estimación del momento adaptativo, de ahí viene el nombre Adam (*Adaptive Moment Estimation*). En la práctica, este es el más conocido y empleado debido a su eficacia y su robustez.

- **Decaída de la tasa de aprendizaje (*Learning rate decay*):** Una de las cosas que puede ayudar a acelerar el algoritmo de aprendizaje es reducir la tasa de aprendizaje a medida que avanzamos en las iteraciones. Como ya dijimos en los apartados previos, una tasa de aprendizaje alta implica que en algunos casos la búsqueda del mínimo en los pasos finales del algoritmo del gradiente se complique. Mediante esta técnica evitamos esto, realizando los primeros pasos del entrenamiento de forma rápida, y consiguiendo más precisión en los pasos

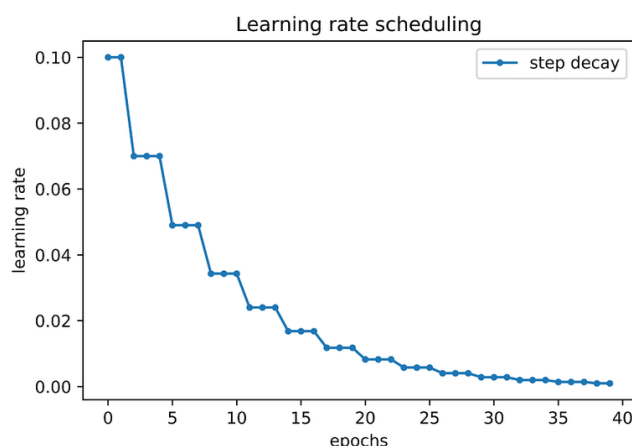


Figura 26. Decaída de la tasa de aprendizaje. Fuente: [ResearchGate](https://www.researchgate.net/publication/312211111)

que se necesite más nivel de detalle. En el siguiente gráfico de la Figura 26, observamos un ejemplo de cómo la tasa de aprendizaje va siendo modificada a medida que avanzamos en las iteraciones. Hay

muchas formas de implementar estas decaídas, todas ellas coinciden en que, en cada iteración, se modifica el valor de  $\alpha$  mediante expresiones preestablecidas (decaída exponencial, decremento por escalones, ...) o de forma manual, aunque menos frecuente.

### Hyperparameter tuning

Los hiperparámetros, al contrario que los parámetros como los pesos o el *bias*, dependen de la configuración que elija el usuario.

El hiperparámetro más importante es la tasa de aprendizaje  $\alpha$ , seguido del número de capas ocultas, el tamaño de los lotes del gradiente o la constante del cálculo de las medias móviles  $\beta$ , y en el último escalón de importancia, el número de capas o la decadencia de la tasa de aprendizaje.

Esta parte se centra en buscar la mejor configuración de hiperparámetros para el modelo. Existen muchas técnicas para elegir los valores óptimos de los hiperparámetros. Nosotros, no entraremos en detalle, pero diremos que las dos técnicas más usadas y que mejor funcionan por un lado es la inicialización aleatoria, y por otra, la reducción del espacio de búsqueda, es decir, coger un valor que funcione de forma eficiente, y realizar una y otra vez la búsqueda con los puntos que se encuentren alrededor de este.

Del mismo modo, existen dos filosofías a la hora de búsqueda de los hiperparámetros óptimos, haciendo analogía con la naturaleza. En nuestro modelo

habrá que reevaluar cada varios meses los hiperparámetros. Para ello, existen estos dos enfoques:

- Panda (*Babysitting one model*): Los pandas tienen un solo hijo, y lo van cuidando mucho centrándose en él (nosotros nos centraremos en un modelo y efectuaremos todas las mejoras y correcciones día a día sobre él).
- Caviar (*Training many models in parallel*): Los peces ponen millones de huevos y hay muchos que se pierden, centrándose solo en los que van bien (haremos lo equivalente con los modelos que den buenos resultados).

Según los costes de CPU y los recursos que dispongamos nos decantaremos ante una filosofía u otra.

### Softmax layer

Hasta ahora solo se ha visto cómo realizar una clasificación binaria. Mediante el clasificador *Softmax*, podemos clasificar un ejemplo a multiclase. La última capa de salida tendrá tantos nodos como clases haya, y cada nodo hará referencia a cada una de estas clases. Como salida, cada nodo devolverá una probabilidad, la cual representará la probabilidad que tiene el ejemplo de pertenecer a esa clase. A la hora de clasificar un ejemplo, adjudicaremos la clase que mayor probabilidad dé en su salida.

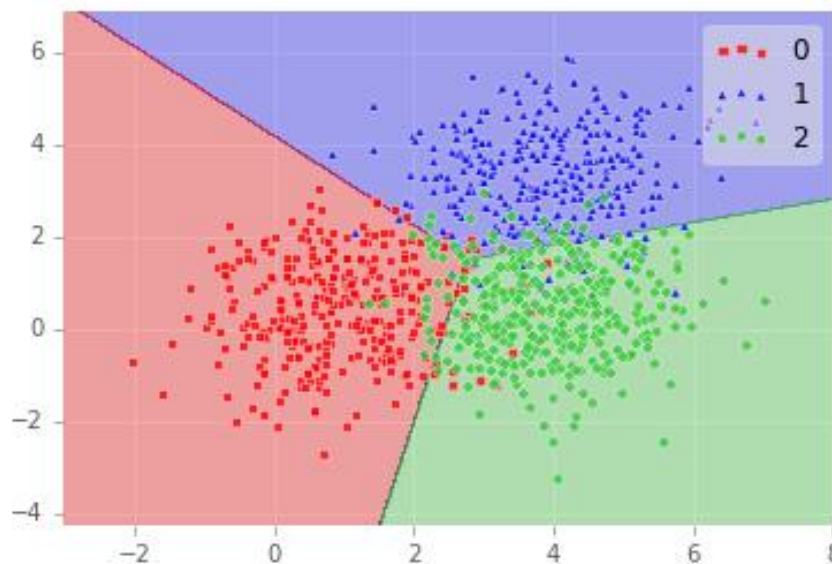


Figura 27. Clasificador Softmax de tres clases. Fuente: [stackoverflow](https://stackoverflow.com)

Por tanto, normalmente, la capa de *Softmax* será la última capa de salida de nuestro modelo. Para comprender mejor su funcionamiento emplearemos un ejemplo: en el ejemplo, deseamos hacer un clasificador, que, a

partir de fotos de animales, obtengamos a qué animal pertenece siendo las clases “perro”, “gato”, “gallina” u “otro”. La red neuronal ha sido previamente entrenada con fotos de estos tipos de animales. La capa *Softmax*, como veremos en el siguiente diagrama, devuelve cuatro probabilidades cada una referente a pertenecer a cada clase.

Aprovechamos también para incluir otra forma de representación de una red neuronal, por bloques, más sencilla ya que obviamos los enlaces entre todos los nodos de la red:

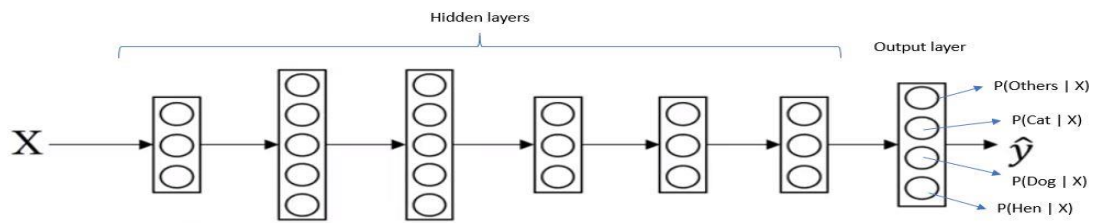


Figura 28. Representación por bloques y Softmax. Fuente: [Coursera](#)

### Receta básica para Machine Learning

Una vez que conocemos multitud de formas de conseguir que nuestro modelo sea más eficiente, como proceso general se establece una receta aplicable a todo modelo de *machine learning*, que describe el proceso de optimización completo, para encontrar el mejor ajuste respecto a *bías* y *varianza*. Este, a modo de guía, nos resume los pasos que tenemos que seguir para mejorar sistemáticamente el rendimiento del modelo. De forma esquemática e iterativa, la emplearemos una vez se haya entrenado el modelo inicial, para primero reducir el *bias*, y posteriormente la *varianza*:

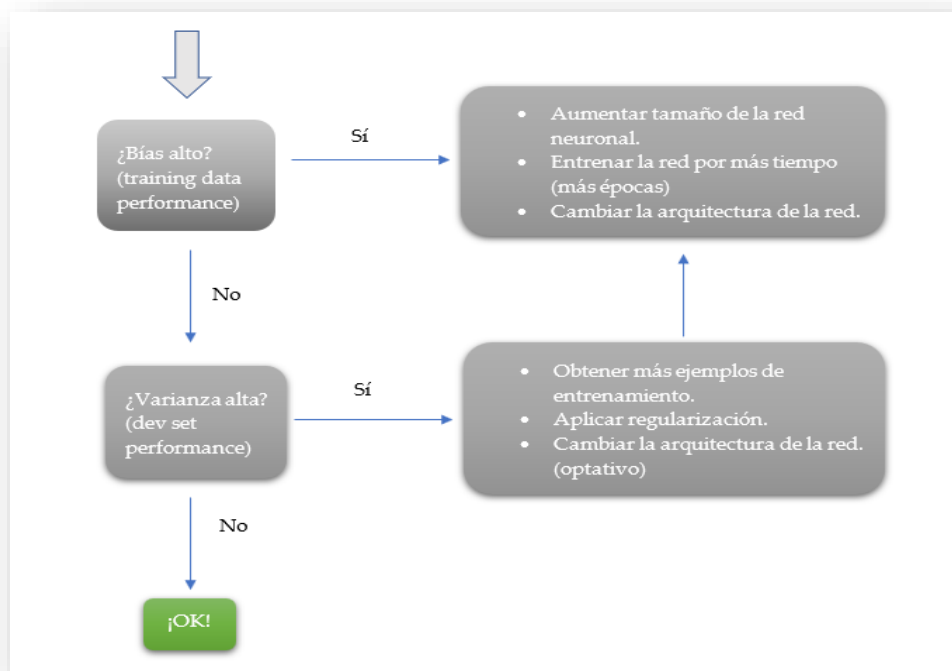


Figura 29. Receta para optimización del modelo.

### 2.2.6 Redes Neuronales Convolucionales

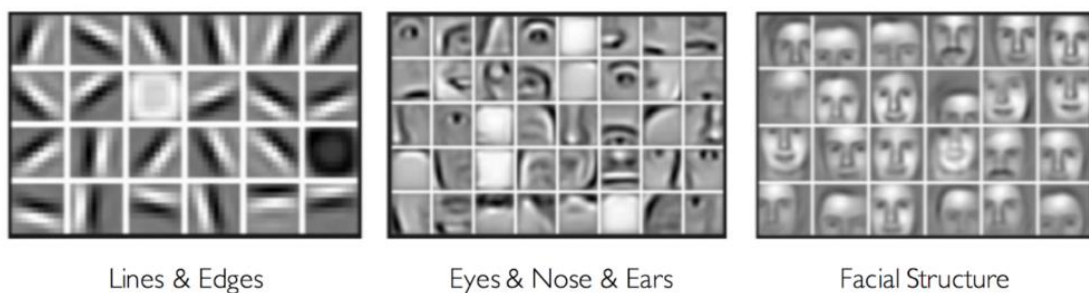
Una vez tengamos una idea general sobre los problemas que son capaces de abordar las redes neuronales, ya podemos entrar más en detalle y analizar los distintos tipos de redes que existen y sus funcionalidades concretas.

#### Introducción

Dentro de las redes neuronales, existen diversos tipos de redes. Cada uno de estos tipos está enfocado a resolver ciertos problemas, o diremos que está enfocado a trabajar con un tipo de datos específico.

Cuando hablamos de una red neuronal que trabaje con imágenes, nos adentramos dentro del universo de las redes neuronales convolucionales, o como también son conocidas, las CNN. Estas, a diferencia de las redes neuronales convencionales, y al igual que realiza el cerebro humano, son capaces de detectar y diferenciar características dentro de las fotos y trabajar con cada una de estas características por separado. Es decir, cada neurona o subred de la CNN se encargará de tratar las características por separado.

De esta forma, las primeras capas de la red serán capaces de tratar características muy específicas como son bordes o cambios de intensidad de los píxeles, y a medida que nos vamos adentrando en las capas ocultas de la misma, pasaremos de reconocer bordes a tratar otras características más generales como son los ojos, boca, pelo... Así hasta llegar a la última capa, donde ya obtendremos la cara completa. Coloquialmente, la red CNN empezará distinguiendo detalles complejos, uniéndolos, consiguiendo cada vez diferenciar aspectos y objetos más generales.



Input---**Shallow Layers**-----**Middle Layers**-----**Deeper Layers** ----> Output

Figura 30. Ejemplo visual de reconocimiento facial en una red convolucional. Fuente: [hackernoon](#)

Otra de las principales ventajas de las CNN respecto a las otras redes es la notoria reducción de tamaño. Esto se consigue gracias a que la CNN trabaja con los mismos pesos para ciertas regiones de la imagen, de este modo y al contrario que en una red

normal, por cada píxel no procesamos un peso, sino que trabajaremos por bloques. Esto, respecto a términos de eficiencia, ahorra mucho coste computacional. Dicho esto, diremos que el principal objetivo, y por ello son utilizadas las CNN en el tratamiento de imágenes, es que a medida que profundizamos en las capas de la red, las características de los datos se van reduciendo, sin perder ningún tipo de información, haciendo que el modelo sea más eficiente. A continuación, veremos cómo se logra estructurar una red convolucional y cómo es su funcionamiento real.

### Estructura y funcionamiento de las CNN

Los datos de entrada para este tipo de redes, al tratarse de imágenes, tendrán forma de matriz. Generalmente y como ya sabemos, los archivos fotográficos se

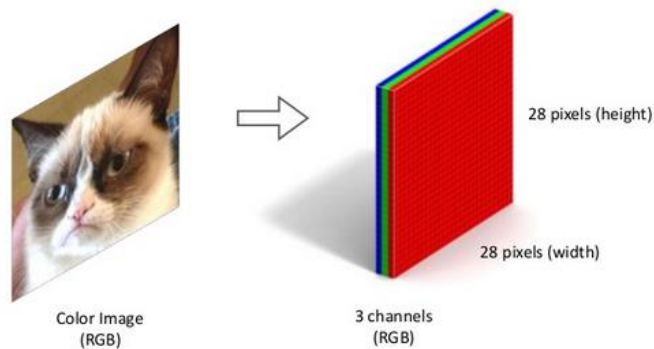


Figura 31. Ejemplo de representación de imagen en forma matricial.  
Fuente: [Coursera](#)

representan mediante tres matrices, una por cada canal de color. Cada matriz contendrá los valores de los píxeles de la imagen. Si cada píxel es tratado como una característica, como ya hemos dicho antes, la cantidad de datos a manipular por cada foto es muy elevado (y cuanto más resolución tenga, más todavía).

Pero partiendo de este punto, ¿cómo conseguimos reducir las características sin perder información útil por el camino? Todo se reduce a emplear distintos filtros a lo largo de las capas de la red. Estos son aplicados sobre la estructura de datos de la capa anterior (siguiendo el ejemplo anterior e inicialmente, tendremos una estructura de  $28 \times 28 \times 3$ ), para conseguir una “predicción” en forma de bloque, de tal manera que en cada capa obtengamos una estructura distinta. En la Figura de la derecha, obtenemos una idea visual de cómo aplicamos estos filtros haciendo un recorrido sobre la estructura de datos.

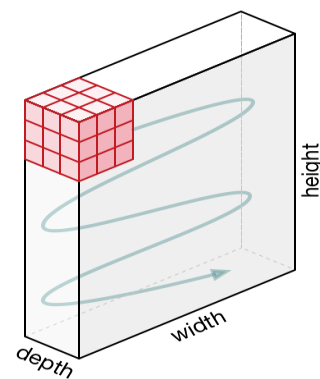


Figura 32. Aplicación de filtro. Fuente: [Coursera](#).



Principalmente, son dos los tipos de filtros (también conocidos como *kernel* o máscaras) presentes en las redes convolucionales. Cada capa de la red se nombrará según el filtro que hayamos empleado en la misma:

- Capas de convolución (*convolutional layers*): Aplican filtros de convolución. Estos filtros se emplean para extraer información concreta de una estructura. Los filtros convolucionales de las primeras capas se encargarán de detectar detalles muy específicos progresivamente hasta detectar objetos más generales. En el ámbito del tratamiento de imágenes, ya existen multitud de filtros predefinidos, más abajo mostramos algunos ejemplos.

Enfoque	Desenfoque	Realce de bordes	Repujado
$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	$\begin{bmatrix} 0 & 0 & 0 \\ -1 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	$\begin{bmatrix} -2 & -1 & 0 \\ -1 & 1 & 1 \\ 0 & 1 & 2 \end{bmatrix}$
Detección de bordes	Filtro de tipo Sobel	Filtro de tipo Sharpen	
$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$	$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} 1 & -2 & 1 \\ -2 & 5 & -2 \\ 1 & -2 & 1 \end{bmatrix}$	
Filtro Norte	Filtro Este	Filtro de tipo Gauss	
$\begin{bmatrix} 1 & 1 & 1 \\ 1 & -2 & 1 \\ -1 & -1 & -1 \end{bmatrix}$	$\begin{bmatrix} -1 & 1 & 1 \\ -1 & -2 & 1 \\ -1 & 1 & 1 \end{bmatrix}$	$\begin{bmatrix} 1 & 2 & 3 & 1 & 1 \\ 2 & 7 & 11 & 7 & 2 \\ 3 & 11 & 17 & 11 & 3 \\ 2 & 7 & 11 & 7 & 2 \\ 1 & 2 & 3 & 2 & 1 \end{bmatrix}$	

Figura 33. Filtros de convolución. Fuente: [UPV](#)

- Filtros *pooling* (*pooling layers*): Aplican filtros de agrupación. El objetivo de estos filtros es únicamente reducir la dimensionalidad de datos de entrada, para permitir luego poder aplicar una máscara convolucional de la forma más eficiente, manteniendo toda la información posible. Son dos tipos de *pooling* los más usados y los que mejores resultados dan: *Max Pooling* (se quedan con el valor máximo de un subconjunto de la estructura de entrada) y *Average Pooling* (devuelven la media del valor de las características analizadas).

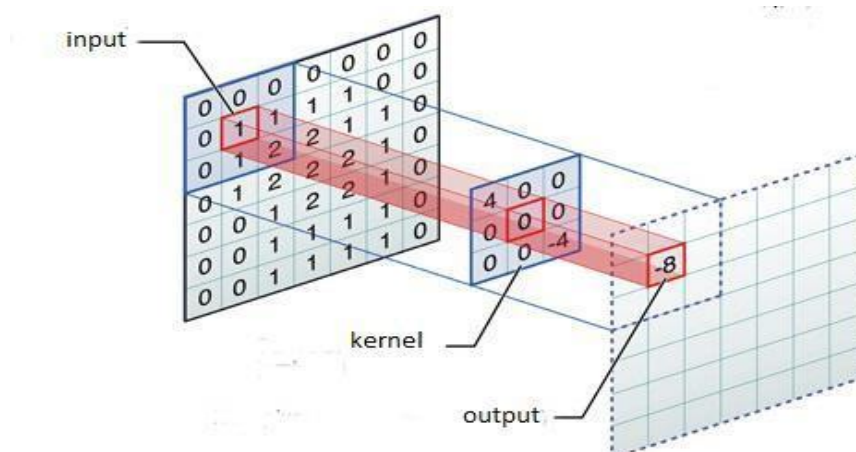


Figura 34. Aplicación del filtro a una sección de la estructura. Fuente: [Coursera](#).

Para estructurar la red, se va alternando el uso de estos filtros a lo largo de las capas de la CNN:

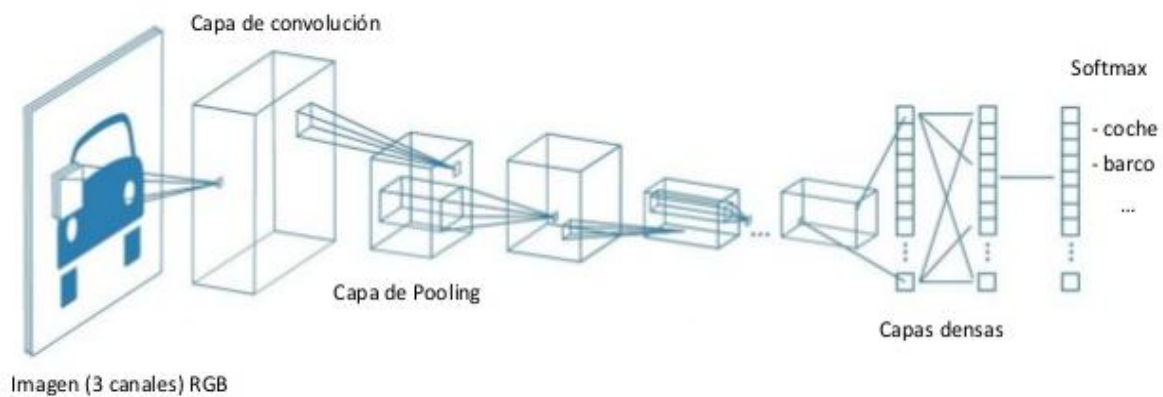


Figura 35. Red CNN. Fuente: [Towards](#)

En la Figura 35 podemos observar una arquitectura simple de una CNN que realiza una clasificación de imágenes. Podemos observar la estructura de los datos y los resultados de aplicar las capas de convolución y *Pooling*.

Finalmente, en las capas finales de las CNN se aplican unas capas que se conocen como capas conectadas (*Fully Connected Layers, FC*). Su funcionamiento es parecido al de las redes neuronales convencionales, conectando todas las características resultantes de la aplicación de los filtros, y así realizar el posterior tratamiento de los datos. En el caso de los clasificadores, se emplearán las anteriormente citadas capas de *Softmax*. En nuestro caso, nos interesará almacenar el vector de características que devuelva la red (a estos vectores se les llama *embeddings*), ya que por cada imagen obtendremos un vector, conservando las características más importantes, y por así decirlo, más maleable y diferenciable.

No entraremos en detalle, pero también tenemos que nombrar que, a la hora de aplicar los filtros, pueden incluirse procesos de *padding*. Estos se realizan cuando hay conflictos entre las dimensiones del filtro y de la estructura de datos. Consiste en aplicar un relleno sobre la estructura de datos para que las dimensiones permitan realizar la operación matemática. Así mismo, también existe la posibilidad de aplicar una técnica de deslizamiento o *stride*. Esta se basa en saltarse algunas posiciones de la estructura a la hora de aplicar el filtro.

Las demás funcionalidades, como el entrenamiento o la evaluación de la red se realizarán de la misma forma que lo hacíamos para las redes neuronales normales.



### Ejemplos de arquitecturas clásicas

Una de las ventajas que ofrece el uso de redes CNN es que existen multitud de arquitecturas predefinidas y que han sido actualizadas a lo largo del tiempo, mejorando su rendimiento. Nombraremos las tres más conocidas y usadas, para tener una idea general de cómo se estructuran de forma real:

- *LeNet-5*: Fue de las primeras arquitecturas CNN funcionales. Se usaba con alrededor de 60000 datos. Hoy en día y con la cantidad de datos que estamos acostumbrados a manejar, resultaría insuficiente.

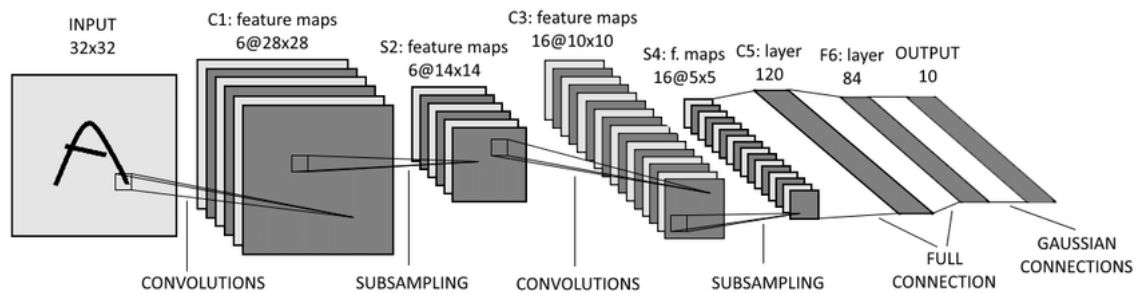


Figura 36. Arquitectura LeNet-5. Fuente: [researchgate](https://researchgate.net/publication/261611147_LeNet-5_Architecture)

- *AlexNet*: Es una evolución de la red anterior, adaptándola para soportar una mayor cantidad de datos (más profunda).

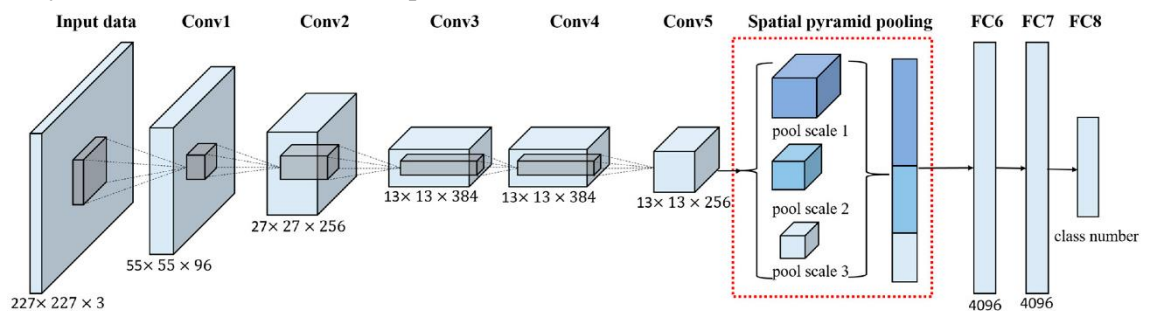


Figura 37. Arquitectura AlexNet. Fuente: [mdpi](https://arxiv.org/pdf/1508.00851v2.pdf)

- *VGG-16*: Esta red se basa en la sencillez de emplear menos hiperparámetros pero más capas de convolución. Luego hablaremos más sobre ella, ya que la emplearemos a lo largo del proyecto.

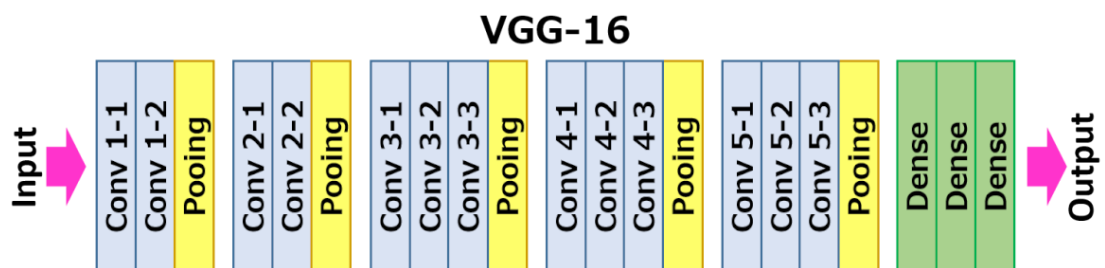
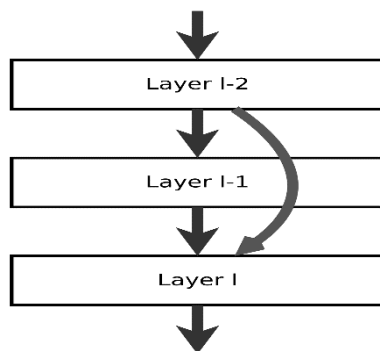


Figura 38. Arquitectura VGG16. Fuente: [Medium](https://medium.com/@vishalsharma23/convolutional-neural-networks-cnns-494480281942)

### ResNets y Redes Inception

Cabe destacar que existen otros tipos de redes, que pueden ser usadas combinadas con las demás arquitecturas.

Por un lado, tenemos las redes residuales (ResNet). Uno de los problemas de las redes



profundas es que a la hora de ser entrenadas el gradiente puede explotar en algún momento determinado rompiendo el proceso entero. Estas redes permiten incluir a la estructura de la red, bloques residuales, permitiendo que, si alguna de las capas falla, el algoritmo pueda avanzar recorriendo la red.

Figura 39. Bloque residual. Fuente: [Wikipedia](https://es.wikipedia.org/wiki/Residual_network)

Por otro lado, existen las redes *Inception*, que a pesar de tener un coste computacional más elevado que el resto de las redes, estas nos dan la posibilidad de no tener que preocuparnos del tamaño de filtro convolucional que deseemos emplear por miedo a que los tamaños no coincidan. Los resultados se van concatenando en módulos, de forma que siempre conservemos las mismas dimensiones.

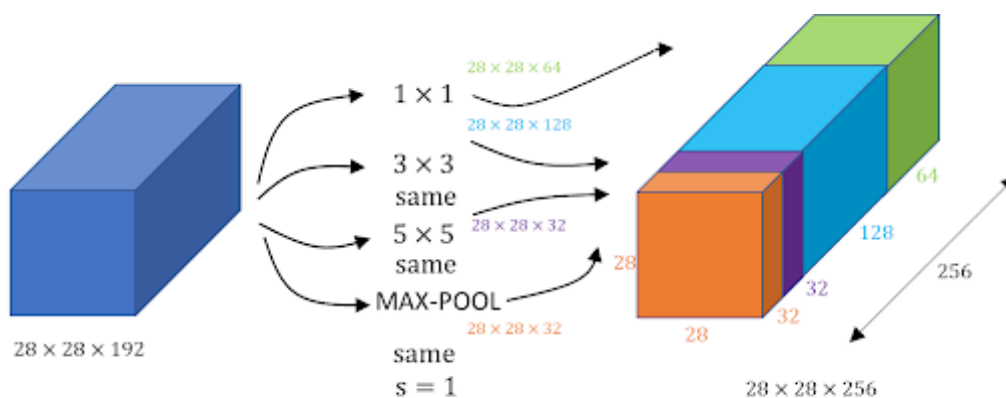


Figura 40. Inception Network. Fuente: [Coursera](https://www.coursera.org/learn/deep-learning-specialization/lecture/29884/inception-network).

### Data augmentation

Muchas veces, por ejemplo, en el problema que vamos a afrontar, se puede dar el caso que el tamaño del conjunto de datos de entrenamiento es muy reducido, lo que provoca que el margen de error a la hora de tratar con imágenes sea muy pequeño.

Por ello, existen multitud de técnicas para aumentar la cantidad de ejemplos de entrenamiento sin modificar la lógica y sin inventarnos nueva información, y de forma

automática. Entre las más empleadas (y las que nosotros emplearemos a lo largo de este proyecto):

- *Mirroring*: Espejar las imágenes.
- *Random Cropping*: Hacer recortes aleatorios sobre las fotos (Este no nos será interesante ya que las fotos ya están recortadas al máximo, y un recorte de un ojo por ejemplo no nos aportará información).
- *Rotation*: Rotación de las imágenes.
- *Shearing*: Transformar inclinando la imagen.
- *Color shifting*: Alterar los valores RGB de la foto (Tampoco haremos uso ya que las imágenes son en blanco y negro).

Más adelante, en la parte donde tratamos las caras de los usuarios, explicaremos con más nivel de detalle el proceso a seguir.

### Reconocimiento facial

El reconocimiento facial es una gama muy amplia dentro del reconocimiento digital, ya que a priori parece un problema sencillo, pero en realidad es más profundo. El reconocimiento facial, o *face recognition*, por definición se trata de una técnica en la cual se realiza una detección de rostro seguida de una detección de vida (para asegurar que no se pueda emplear una fotografía para la suplantación de identidad).

Dentro del reconocimiento facial, se diferencian:

- Verificación facial: Partiendo de una imagen de entrada, una base de datos y una identificación, comprobar si esa persona realmente es quien dice ser (problema 1:1). Ver la persona con ese Id en la base de datos y decir si es o no.
- Reconocimiento facial: Este problema, más complejo que el anterior (1:k), trata de dada una imagen, relacionarlo con una persona. Es decir, recorrer la base de datos entera y asignar esa persona a una clase.

Nosotros, nos enfrentaremos ante un problema de reconocimiento, ya que tenemos que enlazar al usuario con los que tengamos almacenados en la base de datos. Y, si queremos incluir a una nueva persona en la base de datos para que pueda ser reconocida, ¿necesitamos realizar cada vez el proceso de entrenamiento de la red? Obviamente si esto fuese así, en conceptos de complejidad sería inviable y muy poco funcional. Es por eso por lo que la forma de clasificación será distinta.

Uno de los problemas más importantes que existen al realizar un sistema de reconocimiento es que, la mayoría de las veces, los datos de cada persona con los que formamos la base de datos son muy escasos, lo que genera una pérdida de precisión. A lo largo del proyecto veremos cómo afrontarlo.

Llegados a este punto del proyecto, podemos decir que tenemos una vista general y la base de todos los conceptos y herramientas con los que se va a llevar a cabo este trabajo. A medida que vayamos avanzando en él, irán apareciendo nuevos conceptos que serán detallados en cada parte, pero la mayoría de ellos estarán directamente relacionados con los temas que hemos tratado hasta ahora. Así mismo, también cabe destacar que las explicaciones de muchos de los conceptos teóricos que hemos visto en este apartado se completarán a medida que los vayamos utilizando y vayamos avanzando en la realización del proyecto.

## 2.3 Preámbulo

Una vez que tenemos un enfoque general sobre todos los procedimientos teóricos, en este apartado vamos a definir los distintos pasos que se van a desarrollar para completar los objetivos del proyecto. Antes de introducirnos directamente en estudiar cómo se ha llevado a cabo el proceso, con el fin de tener una visión previa y así facilitar la posterior comprensión de cada apartado, vamos a repasar las distintas etapas que se han realizado hasta obtener el sistema de reconocimiento facial.

Como hemos mencionado con anterioridad, el proyecto queda dividido en dos ejes principales: por un lado, la extracción de información y todos los mecanismos que esta engloba, y, por otro lado, el sistema de detección facial.

En la primera etapa del proyecto, describiremos cómo se ha realizado todo el proceso de minería de datos, es decir, a qué procedimientos hemos acudido para obtener las imágenes de los usuarios y los algoritmos que se han utilizado para extraer información de estas (*image captioning*, modelo *bag of words*, detección de rostros...). En definitiva, en esta etapa se definen todos los procedimientos necesarios para completar la estructura de la base de datos (que se detallará más adelante) con sus correspondientes datos. En la segunda etapa y la que termina de dar forma al proceso, explicaremos cómo hemos empleado todos estos datos extraídos para realizar el sistema de reconocimiento facial.

En la siguiente página mostramos dos esquemas generales y visuales de los procesos principales que vamos a realizar, tal y como acabamos de repasar en los párrafos anteriores. En la Figura 41 mostramos un esquema de cómo se realiza la extracción de la información, y en la Figura 42 otro esquema del funcionamiento completo de la aplicación de reconocimiento realizada.

Ahora sí, podemos adentrarnos en la realización del proyecto.

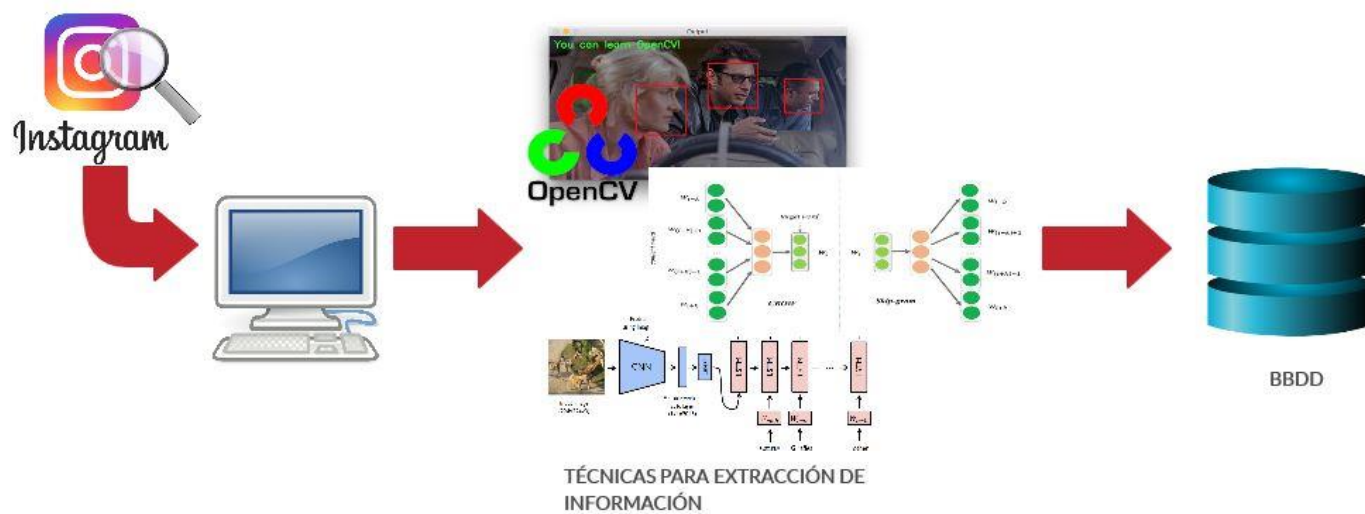


Figura 41. Esquema de la extracción de información.

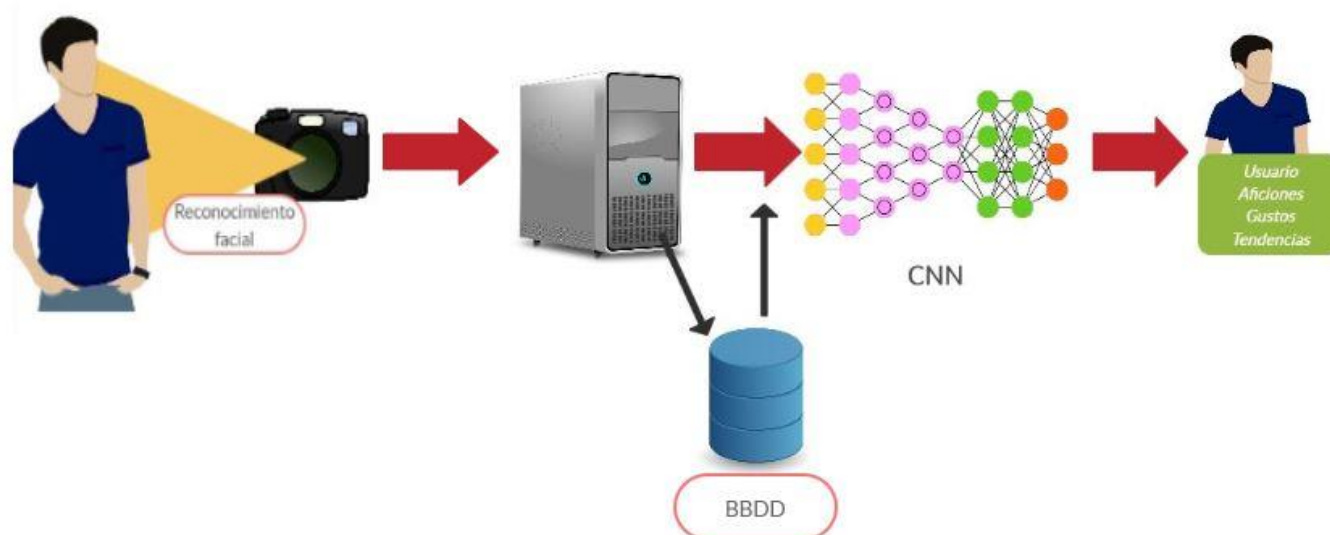


Figura 42. Esquema del funcionamiento de la aplicación.

## 3-Obtención de datos en bruto

En esta primera etapa del proceso, detallaremos cómo se ha realizado la extracción de los datos almacenados en la red social Instagram, así mismo, cómo será la disposición de los datos de nuestra base de datos.

### 3.1 Base de datos

La lógica en la que guardaremos los datos será muy importante ya que, al realizar distintas transformaciones de las imágenes a lo largo del proceso según su fin, estas necesitan estar almacenadas de forma eficiente. En la página 48 (Figura 43), podemos ver un esquema de la estructura y de cómo se van a almacenar los archivos en la base de datos, dentro de cada id de usuario analizado. Esta jerarquía se implementará a través de directorios, que serán creados de forma automática según se vayan realizando las distintas fases del programa.

En el nivel superior, situaremos las carpetas de los distintos usuarios a través de los respectivos ids de la cuenta. En cada una de ellas, almacenaremos:

- Todas las **imágenes** del usuario que hemos descargado a través del proceso de *Web Scraping*. El nombre de cada imagen dependerá del URL donde se encuentre cada una, por lo que no existirán dos con el mismo nombre. El formato para todas ellas es JPG.
- **desc\_id\_instagram.txt**: Fichero donde se almacenarán las descripciones resultantes de aplicar el proceso de *Image Captioning*.
- **rel\_id\_instagram.txt**: Fichero que contiene las palabras más relevantes del usuario. Cuando lleguemos al proceso de relevancia entenderemos en qué consiste.
- **comentNoUser\_id\_instagram.txt**: Fichero que almacenará todos los comentarios que otros usuarios han realizado sobre las publicaciones del usuario analizado.
- **comentUser\_id\_instagram.txt**: Contiene todos los comentarios y respuestas realizadas por el usuario en cuestión.
- **emojis\_id\_instagram.txt**: Almacena los N emoticonos más utilizados por el usuario.
- **tags\_id\_instagram.txt**: Almacena los N *tags* (o tendencias reales) favoritos del usuario.
- **prom\_likes\_id\_instagram.txt**: Almacena, por una parte, el promedio de *likes* o “Me gusta” que ha obtenido el usuario por publicación, y por otra almacena una

sigla que representa el rango de popularidad del perfil. Más adelante profundizaremos sobre este aspecto.

- *titulos\_fotos\_id\_instagram.txt*: Fichero que contiene todos los títulos que el usuario pone a las publicaciones.
- *user\_freq\_id\_instagram.txt*: Guarda los N usuarios más frecuentes o con los que más interactúa el usuario.
- *emb\_id\_instagram.pkl*: Fichero que contiene los *embeddings* de las imágenes completas (*image captioning*).
- La carpeta *caras*, que contiene los recortes de las caras que aparecen en la cuenta del usuario que hemos obtenido mediante el proceso de detección de rostro, más las caras obtenidas tras aplicar la fase de *Data Augmentation*. El fichero *caras\_id\_instagram.pkl* contiene los *embeddings* de todos los rostros.

Toda la ejecución del programa se ciñe en la lógica explicada, por lo que la estructura de esta adquiere gran importancia. A partir de ahora, detallaremos paso a paso el proceso, lo que nos hará comprender mejor el origen de cada archivo mencionado y su futura utilización.

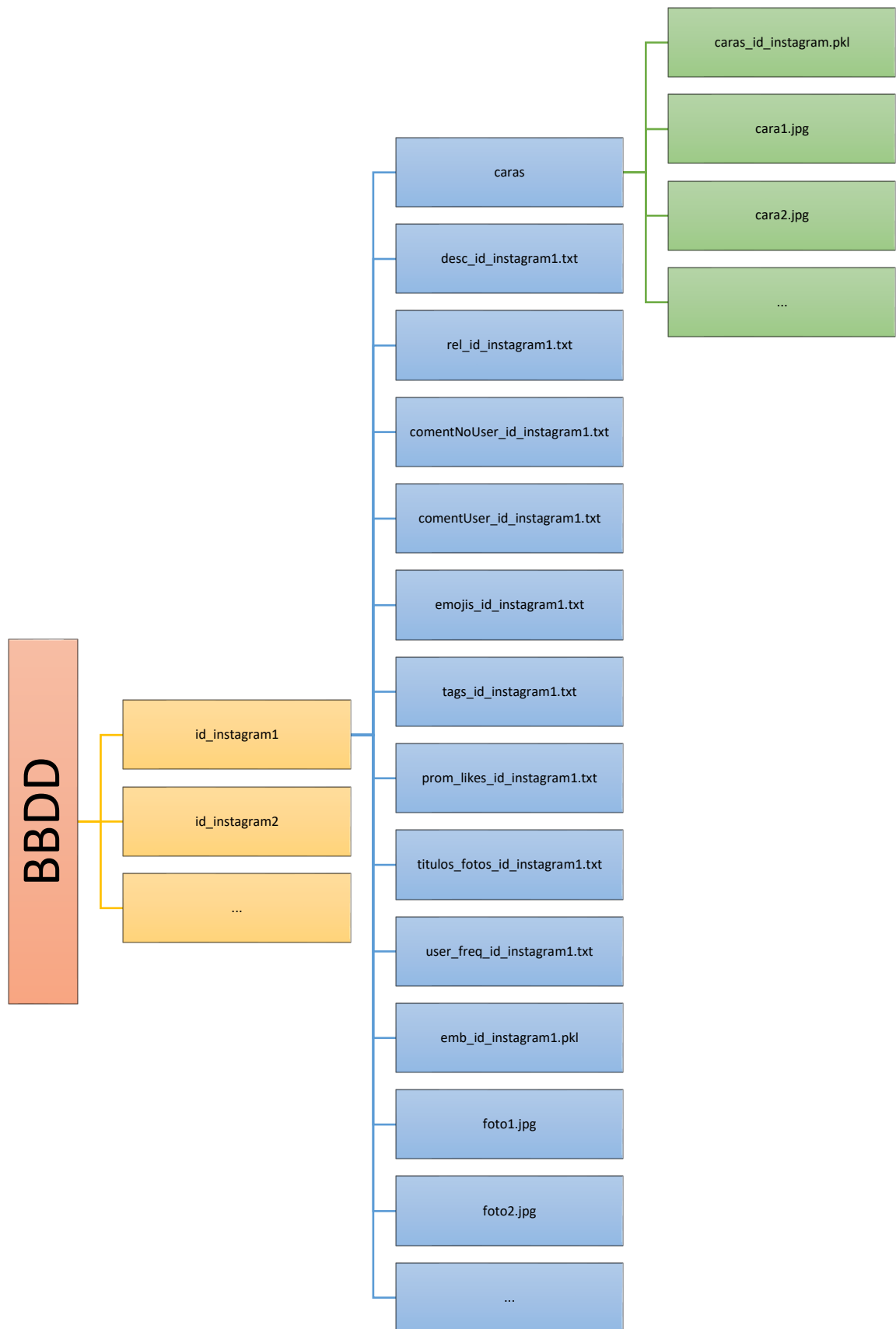


Figura 43: Jerarquía de la base de datos.



## 3.2 Web scraping

El *web scraping* consiste en la obtención masiva de datos almacenados en la red navegando sobre ella, de forma que nos reduzca el mayor tiempo posible dicha labor. La mayoría de *bots* que realizan el scrapeo de datos se basan en recorrer páginas HTML y posteriormente crear las bases de datos. De esta forma, no es necesario descargar uno a uno los datos, en nuestro caso las imágenes, automatizando así el proceso.



Figura 44. Esquema del funcionamiento del Web Scraping. Fuente: [edureka!](https://edureka1.com/)

Muchas veces el *web scraping* está relacionado con notaciones negativas ya que en ocasiones es empleado con objetivo de espionaje o captura de datos, pero no siempre es así, ya que la mayoría de las empresas lo emplean (Google, sin llegar más lejos, para mantener por ejemplo los resultados de las búsquedas de imágenes actualizadas). Otros de los usos más famosos son los agregadores de contenido (recopilar todo el contenido en un único sitio), estudio de sentimientos (por ejemplo, según lo que la gente comenta Twitter), monitorización de datos (con objetivo de ver las tendencias en tiempo real de la sociedad) ...

A pesar de todo, en muchas páginas web hoy en día podemos ver los ya conocidos *captchas*, los cuales se emplean para evitar el uso de *bots* automáticos como pueden ser los *scrapers*.

### 3.2.1 Legalidad

Existe un debate legal y ético con la extracción automática de datos, y aún más en si los datos pertenecen a personas, por lo que estamos invadiendo privacidad y vulnerando de cierto modo los derechos de cada uno. En nuestro caso particular, al emplear los datos con fines de investigación, no nos estamos lucrando de ello como lo podría hacer una empresa, por lo tanto, no habría problema. Más incluso si obtenemos el consentimiento previo de las personas propietarias de los datos en cuestión.

Respecto al scrapeo de datos de las páginas web, cada página web posee sus condiciones legales por lo que debemos cumplirlas. En estos casos estaríamos vulnerando la ley por lo que debemos ser cuidadosos con la fuente de origen de donde escogemos los datos.

Si hablamos de legislación, en España diremos que realizar esta práctica no es ilegal, pero lo que no está permitido es violar los derechos de autor empleando esta técnica (Ley de Propiedad Intelectual: *La propiedad intelectual de una obra literaria, artística o científica corresponde al autor por el solo hecho de su creación*).

### 3.2.2 Proceso de extracción

Para la obtención de las imágenes almacenadas en Instagram emplearemos un *Scraper* preconstruido (<https://github.com/rarcega/instagram-scraper>). Este ha sido elaborado en lenguaje Python, y se instala mediante PyPI (Python Package Index), herramienta de Python que realiza un catálogo de paquetes de terceros, que permite la instalación de aplicaciones de código abierto.

Una vez descargado el programa, este se puede instalar mediante el comando *pip install Instagram-scraper*. Esto nos creará en el directorio de *scripts* donde ha sido instalado previamente Python en nuestro ordenador el ejecutable *exe*.

Para ejecutar el programa mediante la consola de comandos bastará con situarnos en el directorio donde está dicho *script* y ejecutar *Instagram-scraper* seguido de la opción que nos interese (con el comando *Instagram-scraper -help* obtenemos un listado de las posibles opciones del programa).

A nosotros en concreto nos interesa la opción de *instagram-scraper <username> -u <your username> -p <your password> -d carpeta-destino*.

Previamente y para hacer más sencillo el proceso de extracción, hemos creado una cuenta de Instagram vacía, de nombre “tfgupnapeio8”, que únicamente tenga de seguidores las cuentas con las que vamos a trabajar. Inicialmente solo tengo de seguidor mi cuenta personal para comprobar su correcto funcionamiento.

Ahora bien, ejecutando el comando *Instagram-scraper usuario\_a\_scrapear -u tfgupnapeio8 -p tfgupnapeio123 -d ruta\_destino* obtendremos una descarga automática de todas las fotos del usuario almacenadas en una carpeta con el nombre del usuario.

```

C:\Users\Peio\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.7_qbz5n2kfra8p0\LocalCache\local-packages\Python3
7\Scripts>instagram-scraper username peio8 -u tfgupnapeio8 -p tfgupnapeio123 -d C:\Users\Peio\Desktop\Fotos_scrap
Searching username for profile pic: 100% 1/1 [00:00<00:00, 101.70 images/s]
Searching username for stories: 0 media [00:00, ? media/s]
Searching username for posts: 0 media [00:00, ? media/s]
Downloading: 100%|#####| 1/1 [00:00<00:00, 2.32it/s]
Searching peio8 for profile pic: 100% 1/1 [00:00<00:00, 497.43 images/s]
Searching peio8 for stories: 0 media [00:00, ? media/s]
Searching peio8 for posts: 15 media [00:05, 2.96 media/s]
Downloading: 100%|#####| 16/16 [00:43<00:00, 2.71s/it]

```

Figura 45. Ejecución del Scraper.

En la anterior captura (Figura 45) tenemos un ejemplo visual de la ejecución del programa mediante la consola. En él ejecutamos el programa en la ubicación donde se ha instalado el *script* almacenando los datos en la carpeta “Fotos\_scrap”.

Una vez comprobado su funcionamiento, necesitamos incluir esta herramienta a nuestro proyecto. Existen dos formas de ejecutar un *script* en el lenguaje Python. La primera, iniciando la sentencia mediante “!”, es decir, *!instagram-scraper*. De esta forma le estamos indicando al compilador que se trata de una acción externa como es este caso, la ejecución de un *script*. Pero esta forma de ejecución no nos será útil ya que no podemos pasar variables del programa a la propia ejecución (nos interesa que los nombres de usuario estén en una variable). Es por eso por lo que emplearemos la herramienta *subprocess*. Esta es la forma que tiene Python de ejecutar un proceso ajeno al programa. Crearemos el subproceso que ejecute el comando *instagram-scraper users -u tfgupnapeio8 -p tfgupnapeio123 -t image story-image --comments -n -d C:\\Users\\Peio\\Desktop\\TFG\\APP\_GENERAL\\BBDD\\*, en donde “users” hace referencia a la variable que contenga los nombres de usuario. Como podemos apreciar, hemos introducido nuevas opciones a la instrucción:

- “-t”: Indica el tipo de archivos que nos queremos descargar del usuario, es decir, imágenes, vídeos o *stories* (*image*, *video*, *story*).
- “-n”: Activa el *flag* que hace que se cree un directorio por cada usuario, de esta manera realizamos automáticamente la estructura de la base de datos.
- “--comments”: Almacena todos los comentarios de esa cuenta en un archivo *json*.
- “users”: Será la variable que contenga la lista de usuarios en formato “user1,user2,user3,...”.

De esta manera, introduciremos en el programa una serie de nombres de usuario con los que vamos a trabajar, y este automáticamente descargará los datos y los almacenará creando así la estructura de la base de datos general.

A partir de aquí, ya tenemos la fuente principal de los datos, podemos empezar a manipularlos y trabajar con ellos.

## 4-Generación de la base de datos de caras

En esta etapa del proyecto realizaremos la criba de imágenes con las que vamos a trabajar. Por un lado, emplearemos únicamente los rostros de las imágenes para el posterior reconocimiento facial, y, por otro lado, usaremos las imágenes completas para la generación de descripciones (en el siguiente capítulo). Una vez hayamos recortado los rostros, nos quedaremos únicamente con aquellos que nos sean de utilidad.

### 4.1 Detección de rostro

En esta primera parte del proceso, nos basaremos en el empleo la librería [OpenCV](#), que ofrece amplio soporte a problemas de visión artificial y procesamiento de imágenes, como es en este caso la detección de rostros.

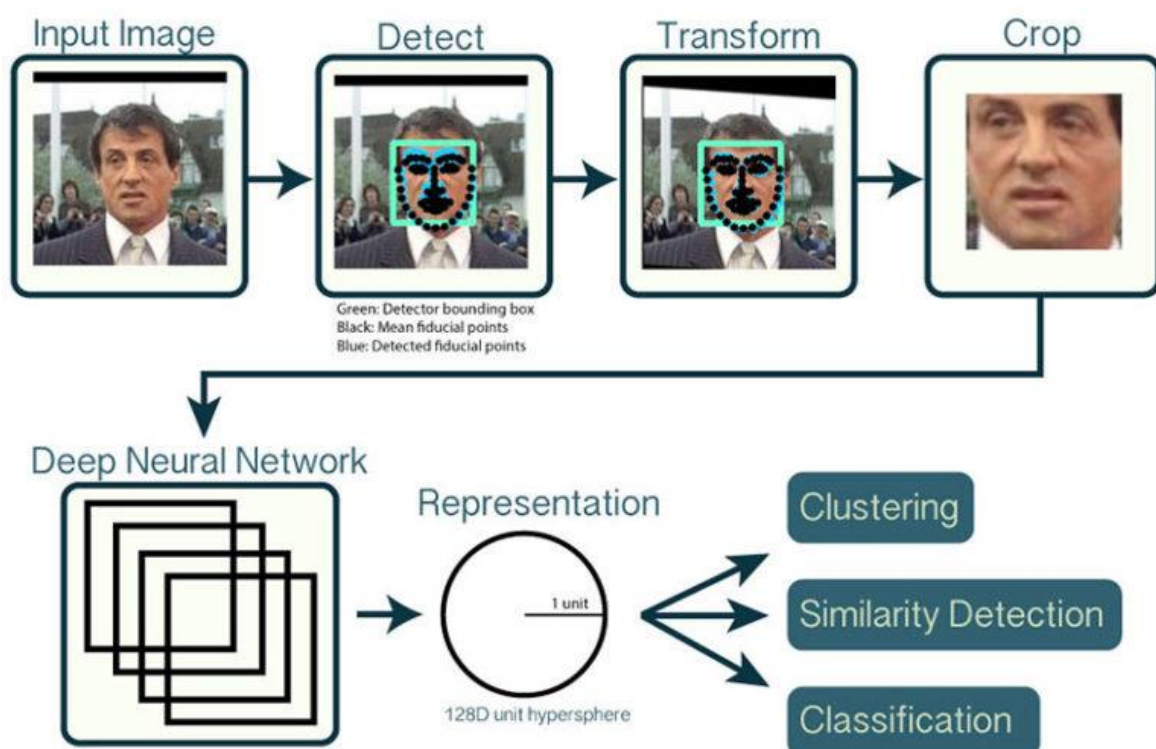


Figura 46. Etapas del reconocimiento facial. Fuente: [pyimagesearch](#)

En la imagen anterior de la Figura 46, se muestra el proceso completo que vamos a realizar en el reconocimiento. En esta parte del trabajo nos centraremos en la parte

superior del diagrama, identificando las fotos que contengan caras y hasta recortar el rostro, dejando para más adelante el trato con la red neuronal.

Dicho esto, pasamos a procesar las imágenes, para así detectar el mayor número de caras posible. Cuantas más caras del usuario identifiquemos, mayor precisión obtendremos a la hora de reconocerlo.

Para la realización de este proceso nos basaremos en emplear filtros de *Haar* en cascada, cuya implementación es proporcionada por *OpenCV*, pero antes, explicaremos los fundamentos del algoritmo en los que se basa dicha implementación.

#### 4.1.1 Clasificador Haar

Este clasificador es muy usado debido a la eficacia y rapidez a la hora de reconocer e identificar objetos en visión artificial. Se basa en aplicar filtros de *Haar* en las imágenes de tal forma que busca patrones de diferencias de intensidades como se pueden dar en los ojos o distintas zonas.

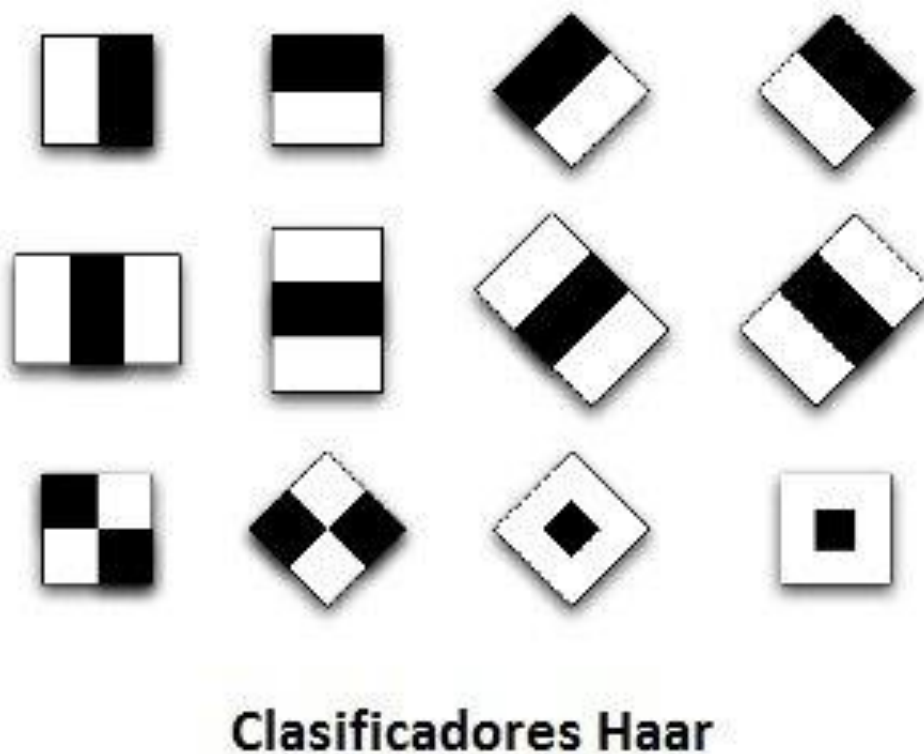


Figura 47. Modelos de filtros Haar empleados en reconocimiento de objetos. Fuente: [unipython](http://unipython.com)

El algoritmo crea subdivisiones de rectángulos en la propia imagen original (imagen integral), comparando las características extraídas de cada uno. Finalmente, el algoritmo buscará en la imagen combinaciones de estos bloques, detectando el objeto si las características coinciden.



Figura 48. Aplicación de los bloques sobre un rostro. Fuente: [robologs.net](http://robologs.net)

Estos clasificadores necesitan ser previamente entrenados con cientos de imágenes positivas (donde aparece el objeto a detectar, en nuestro caso, el rostro) y negativas (no aparece el objeto). El sistema tiene un porcentaje de aciertos bastante alto, aunque su éxito también dependerá de la calidad de imagen (contraste, brillo, ...).

#### 4.1.2 Implementación del detector

*OpenCV*, además de aportar las funciones necesarias para realizar una detección de objetos, también pone a disposición ficheros "XML" que contienen los modelos previamente entrenados para poder detectar ciertos objetos sencillamente. Nosotros nos aprovecharemos de la implementación del modelo que reconoce los rostros para nuestra propia implementación, pero, además, podemos hacer uso de modelos que detectan los ojos, sonrisa, gafas de sol...



Para ello, simplemente descargamos el archivo de código abierto “haarcascade\_frontalface\_default.xml”, que contiene todas las herramientas necesarias para el proceso y lo incluimos en la ruta del programa general.

Una vez importadas las librerías de *OpenCV* (cv2), instanciaremos el objeto del clasificador mediante *CascadeClassifier*, pasando como argumento el fichero *xml* descargado. Hecho esto, ya dispondremos de un detector de rostros en imágenes.

Para detectar los rostros, bastará con llamar a la función *detectMultiScale* y pasarle la imagen a detectar (previamente pasada a escala de grises para que sea más eficiente y precisa), junto con unas *flags* previamente definidas. El resultado de la llamada a la función es una lista con las coordenadas de los rectángulos que encierran los rostros. Para observar el correcto funcionamiento, hemos dibujado sobre algunas imágenes propias los rostros detectados mediante la librería *matplotlib*:

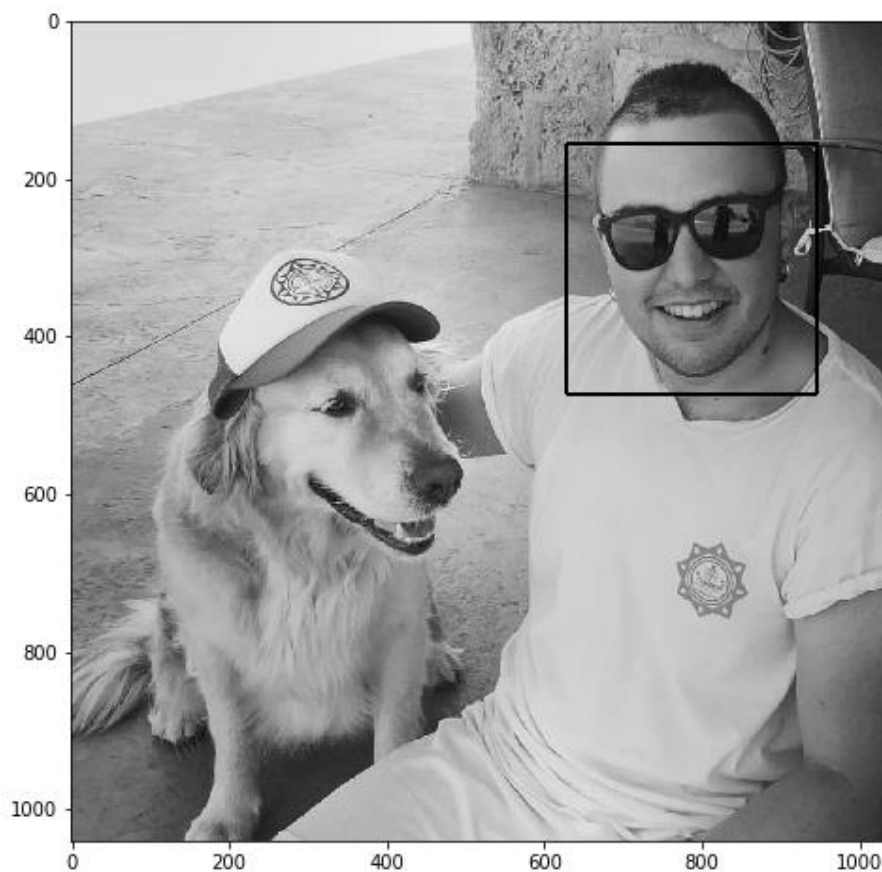


Figura 49. Resultado de la detección facial en una foto del usuario.

En el ejemplo anterior (Figura 49) podemos comprobar que la implementación funciona correctamente incluso en casos complicados, ya que al aparecer con gafas los patrones de la cara son diferentes. A pesar de esto y cómo podemos observar en la siguiente Figura 50, las caras que aparecen incompletas o recortadas son más complicadas de reconocer porque no conservan todas las características. Esto no lo considero importante, ya que en la mayoría de los casos donde aparece más de una cara, la cara del usuario a reconocer que es la que nos interesa, es muy improbable que sea la que aparezca incompleta.

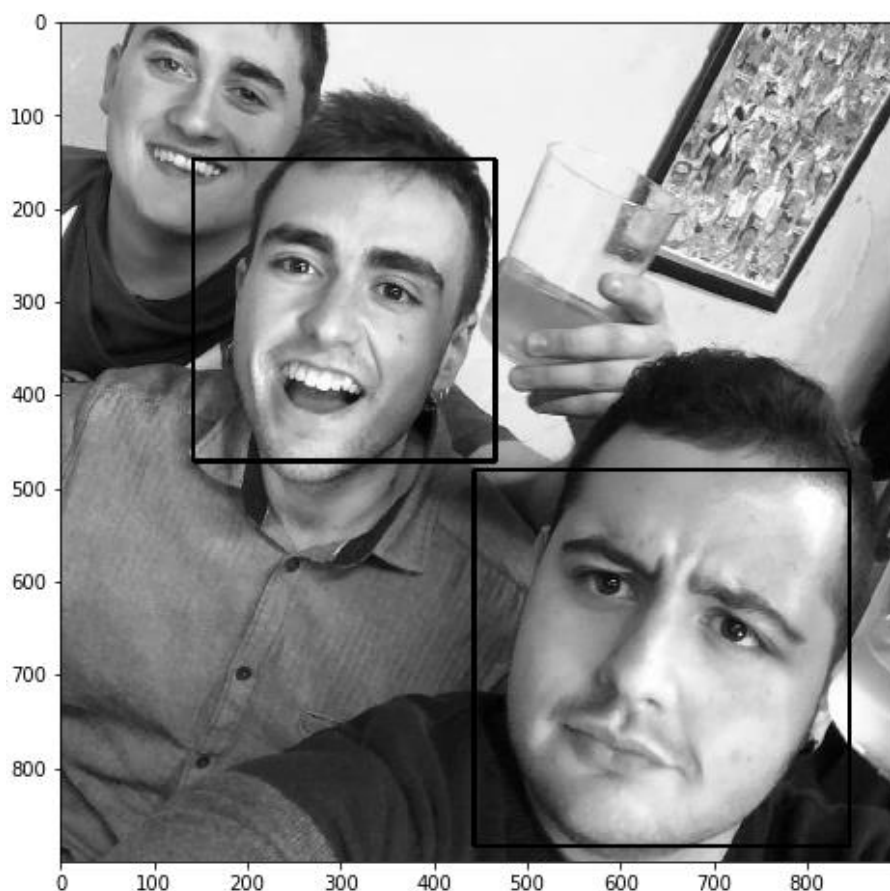


Figura 50. Otro ejemplo de detección facial..

Una vez comprobado el funcionamiento, pasaremos a aplicar el programa sobre un conjunto de imágenes, es decir, vamos a reconocer todas las caras que aparecen en todas las fotos (descargadas con el *scraper*) del usuario a analizar.

Para ello, creamos una carpeta llamada “caras” dentro de la carpeta donde están las fotos del usuario, en donde se almacenarán todos los recortes de los rostros que aparecen en las fotos. Primero, se almacenarán todas revueltas (las del usuario



mezcladas con caras de personas ajenas que parecen en las fotos), más adelante veremos cómo seleccionamos únicamente las caras del usuario.

El programa principal se compone de un bucle, que recorre todas las imágenes realizando los pasos citados anteriormente una a una (convertir a escala de grises y lanzar el método *detectMultiScale*). Por cada imagen obtenemos un listado de coordenadas. Para almacenar únicamente el recorte de la cara, basta con seleccionar sobre la imagen general el área formada por el rectángulo de las coordenadas y guardarlo mediante *imwrite* en un fichero.



Figura 51. Almacenamiento de las imágenes con las caras resultantes.

Mostramos un recorte de cómo quedarían almacenadas una parte de las caras resultantes tras ejecutar la aplicación, en la Figura 51. Ahora, nos interesan solo las caras donde aparece el usuario para poder entrenar la red únicamente con su rostro y reconocerlo más adelante. Por ello necesitamos descartar las demás caras detectadas, pero ¿cómo sabemos qué cara es la del usuario entre todas ellas? A continuación, explicaremos cómo realizar esta selección.

## 4.2 Clasificación de caras

Una vez hayamos obtenido todas las caras que aparecen en las fotos, queremos quedarnos solamente con los rostros del usuario, los cuales emplearemos para entrenar la red convolucional de reconocimiento facial y así obtener una mayor eficacia.

La dificultad de este apartado consiste en como asignar una cara al usuario que estamos tratando, es decir, cuáles de todas las caras recortadas pertenecen a la persona en cuestión.

En nuestro caso, el porcentaje de caras recortadas pertenecientes al usuario oscila en torno al 60%, es por ello por lo que supondremos por obviedad que la clase mayoritaria pertenecerá al usuario.

El problema viene cuando un usuario de Instagram no contiene fotos en las que aparece su rostro, por lo que la predicción de clase mayoritaria fallará, asignando mal el rostro del usuario. A pesar de todo, como creemos que este fallo (en muy pocas cuentas se da el caso de que el rostro del usuario es el que menos aparece) es muy inusual, nos ceñiremos en nuestra suposición de asignar el rostro mayoritario al usuario en cuestión.

### 4.2.1 *Procesamiento de imágenes*

Antes de aplicar cualquier algoritmo sobre las imágenes, estas necesitan ser previamente procesadas para que sean manipulables. Es decir, necesitamos procesar estas imágenes mediante una red convolucional para conseguir extraer todas las características relevantes (más adelante, en el apartado de *Image Captioning* emplearemos otro modelo distinto para realizar esta transformación).

#### *CNN y generación de embeddings*

Basándonos en todo lo que hemos tratado en los apartados teóricos sobre redes neuronales y redes convolucionales, a continuación, detallaremos la forma de utilización de este concepto en nuestro proyecto.

El objetivo principal de esta fase dentro del procesamiento de imágenes consistirá en procesar todos los datos de entrada (en este caso los rostros de cada usuario resultantes de aplicar la detección de rostros) mediante una red convolucional (CNN). Esta red será la encargada de, pasada una imagen de entrada, extraer todas las características del rostro, devolviéndolas en forma de vector, o también llamado *embedding*.

De este modo, las imágenes, que actualmente las almacenamos en la base de datos, siguiendo la estructura de esta, en forma de archivo físico (a nivel de programación en matrices bidimensionales, ya que están en escala de grises), pasarán a

ser procesadas a través del modelo obteniendo un conjunto de *embeddings* como salida. Estos vectores, por un lado, a nivel físico serán almacenados en un fichero dentro de la ruta de cada usuario. De esta forma nos ahorramos el tener que procesar todos los rostros cada vez que vayamos a ejecutar el programa, haciendo uso de una función que carga masivamente el contenido de los ficheros (más eficiente). Y, por otro lado, a nivel de computación, los almacenará en un diccionario (los nombres de usuario las claves y los *embeddings* los valores).

### El modelo: FaceNet

Para poder procesar los datos y extraer las características, necesitamos previamente definir un modelo y realizar su correspondiente aprendizaje. Para que el reconocimiento facial funcione adecuadamente, tal y como hemos visto en la teoría, juega un papel muy importante la definición del modelo y que este generalice bien los datos. Para conseguir un ajuste de parámetros óptimo, es necesario un conjunto de datos de entrenamiento lo suficientemente grande y completo, y aparte, también es necesario disponer de los recursos y el tiempo suficientes como para realizar este entrenamiento del modelo. Ante la carencia de esto último, vamos a hacer uso de un modelo muy popular, que se emplea en la mayoría de los problemas de reconocimiento facial de IA actualmente, el cual está previamente entrenado y su integración es muy sencilla.

Se trata de un modelo de red convolucional ([Facenet](#)). Mediante este, las imágenes de entrada (rostros) se descomponen en las principales características del rostro (ojos, nariz, boca...). Para emplear el modelo en el proyecto, bastará con añadir el archivo de *Facenet* en la ruta de nuestro proyecto (archivo que contiene los pesos del modelo pre-entrenado) y cargarlo con la función *load\_model* que aporta *keras*. Una vez hecho esto, ya podemos emplearlo mediante *predict* y la imagen de entrada.

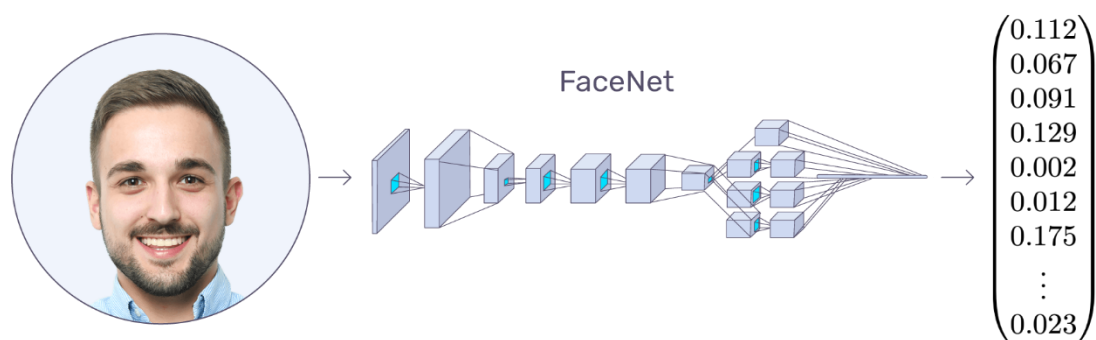


Figura 52. Esquema FaceNet. Fuente: [arsfutura](#).

### Generación de los *embeddings*

El primer paso que realiza nuestro programa será cargar el modelo citado tal y como hemos explicado e ir recorriendo la carpeta en donde se almacenan las caras, pasándolas una a una por el mismo. De ahí obtendremos todas las codificaciones de las imágenes en forma vectorial (*embedding*), las cuales almacenaremos en un diccionario junto al nombre o id de cada imagen. En la Figura 53 mostramos otro ejemplo visual de este procedimiento.

Los *embeddings* serán vectores de dimensión 4096, en donde cada posición del vector hará referencia a una característica concreta de la imagen procesada, ponderando esa característica según esté definido el modelo. En nuestro modelo de FaceNet, estas ponderaciones estarán enfocadas en resaltar relieves y detalles los cuales hacen que un rostro sea distinguible de otro. A donde voy es que, obviamente, si hubiéramos hecho una red enfocada a la detección de automóviles en imágenes, la ponderación de los pesos y la extracción de características sería completamente distinta.

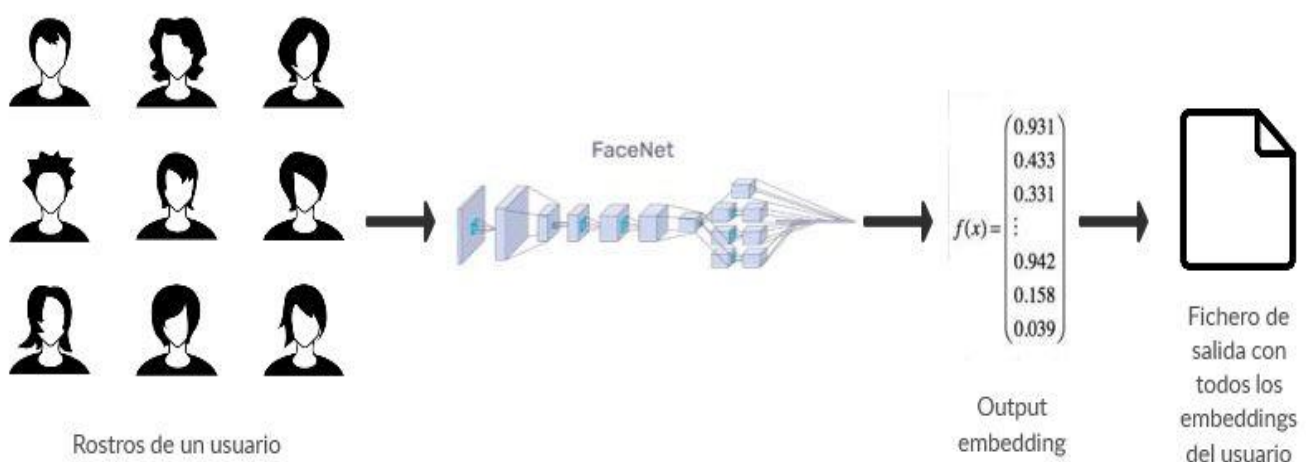


Figura 53. Generación de *embeddings* de un usuario.

#### 4.2.2 PCA (Principal Component Analysis)

Como bien conocemos, existe un algoritmo llamado PCA, el cual permite realizar una reducción de dimensiones de los datos haciéndolos más distinguibles entre sí. Esto, matemáticamente se basa en el cálculo de la matriz de correlaciones, de tal manera que se aplica una transformación lineal sobre el conjunto original de datos de entrada.

La librería *Sklearn* nos proporciona una funcionalidad llamada PCA, que lo que hace es realizar automáticamente una reducción de características implementando el algoritmo, pudiendo reducir así el tamaño de características de entrada de 4096 (en nuestro caso) al número de componentes que deseemos. Nosotros, hemos querido aplicar esta técnica para visualizar los datos en gráficas tridimensionales (es decir, reducimos el conjunto de datos a tres componentes), para observar en el espacio los distintos grupos de imágenes parecidas que se generan por cada usuario. Este proceso únicamente lo realizamos para visualizar y entender mejor los datos, ya que, para realizar la clasificación de estos, emplearemos todas las características de cada *embedding*.

Hemos aplicado este proceso a diferentes usuarios de la base de datos para ver la representación gráfica de los rostros de cada uno, generando como antes hemos dicho, tres componentes principales por cada imagen. De este modo y como observamos en la Figura 54, cada gráfica hace referencia a un usuario de la base de datos, y los puntos son la disposición espacial de los rostros recortados de cada uno de ellos. Podemos ver de forma más o menos clara que en algunos, se generan grupos de datos bastante distinguibles entre sí.

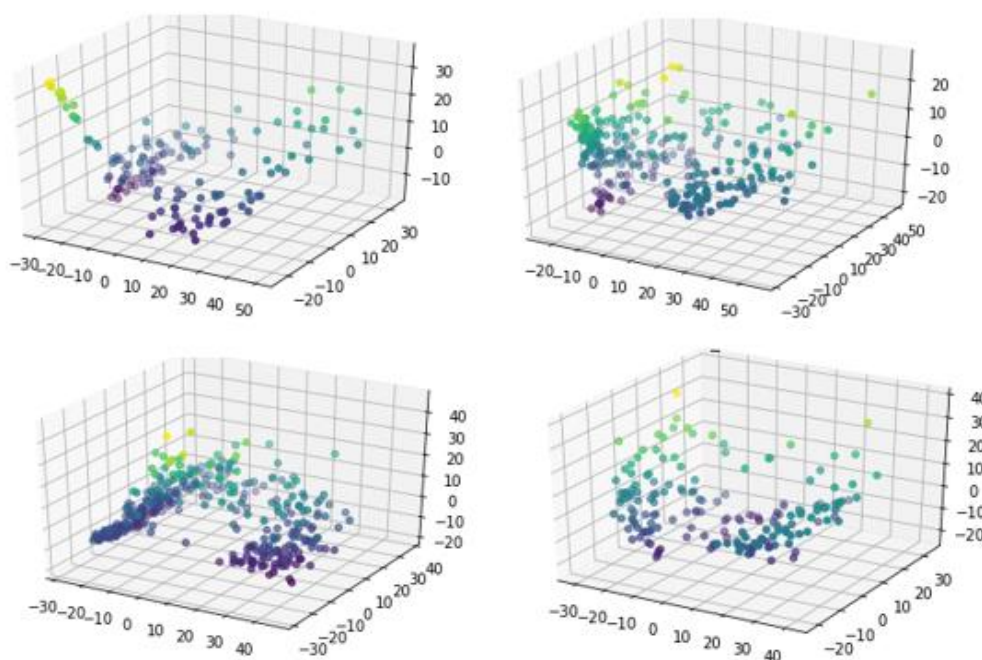


Figura 54. Modelos de recortes de caras de distintos usuarios.

### 4.2.3 Reconociendo la cara del usuario, Algoritmo K-means

Ahora, el paso a realizar es saber reconocer cuál de todas las caras recortadas por cada perfil de Instagram es la del usuario propietario de la cuenta. Partiendo de la base, como ya hemos dicho antes, que suponemos que la cara mayoritaria en el modelo de datos será la del usuario, realizaremos el algoritmo K-means para clasificar los datos que hemos procesado hasta ahora, y descartar los datos innecesarios (caras mal detectadas y caras que no son las del usuario que estamos analizando). Trataremos de resumir brevemente el funcionamiento del algoritmo para entrar en contexto.

#### Funcionamiento de K-means

K-means es un algoritmo no supervisado de aprendizaje de Clustering. El objetivo de este, al tratar con datos sin etiquetar, es de encontrar patrones en los datos o posibles clases que hagan a los ejemplos del conjunto de datos diferenciables los unos de los otros. Dicho de otra manera, el algoritmo tratará de identificar unos clústeres o centros, e ira asignando los datos a los centros que más se parecen, obteniendo datos parecidos entre sí con este criterio.

Es por esto por lo que decimos que se trata de un algoritmo iterativo. Este calcula los centros (ajustándolos en cada iteración), de tal forma que la suma total de las distancias de los ejemplos respecto al centro (o clase) a la que se le asigna sea la menor posible.

Más concretamente, el algoritmo sigue los siguientes pasos:

- 1- **Asignación de centroides:** Este paso consiste en recorrer el conjunto de datos y asignar a cada ejemplo el clúster más cercano. Como métrica se pueden emplear distintos tipos de distancias (Manhatan, Euclídea, ...). Nosotros emplearemos la distancia Euclídea:

$$\operatorname{argmin}_{c_i \in C} \operatorname{dist}(c_i, x)^2$$

Figura 55. Expresión de la distancia Euclídea.

- 2- **Actualización de centroides:** En este paso se realiza el proceso iterativo, de tal manera que se recalculan en cada iteración minimizando la distancia media de los ejemplos de su clase, volviendo a asignar los ejemplos si fuera necesario. El algoritmo itera hasta alcanzar un criterio de parada (si se alcanza el número



máximo de iteraciones, la distancia media se minimiza por debajo de un umbral o si la asignación de ejemplos no varía más).

$$c_i = \frac{1}{|S_i|} \sum_{x_i \in S_i} x_i$$

Figura 56. Expresión del cálculo del peso de los centroides.

- 3- Elección del valor de K (número de centroides):** El valor que toma K también influirá sobre el rendimiento del programa, ya que necesitamos obtener el número de clases que haga más distinguibles los ejemplos entre sí. Inicialmente el valor es predefinido por el usuario, y la única forma de conseguir la mejor configuración posible para nuestro modelo es probar con distintos valores que puede adoptar K. Nos quedaremos entonces con aquella K que produzca la mejor configuración deseada para el modelo (en nuestro caso, el mayor número de caras del usuario posibles y el menor número de rostros ajenos en el mismo clúster). Si esto lo representamos gráficamente, nos encontramos ante una típica función, denominada “función codo”, en donde el objetivo es encontrar el punto donde se dé la variación máxima de pendiente y a la vez el valor de la función sea el más bajo posible.

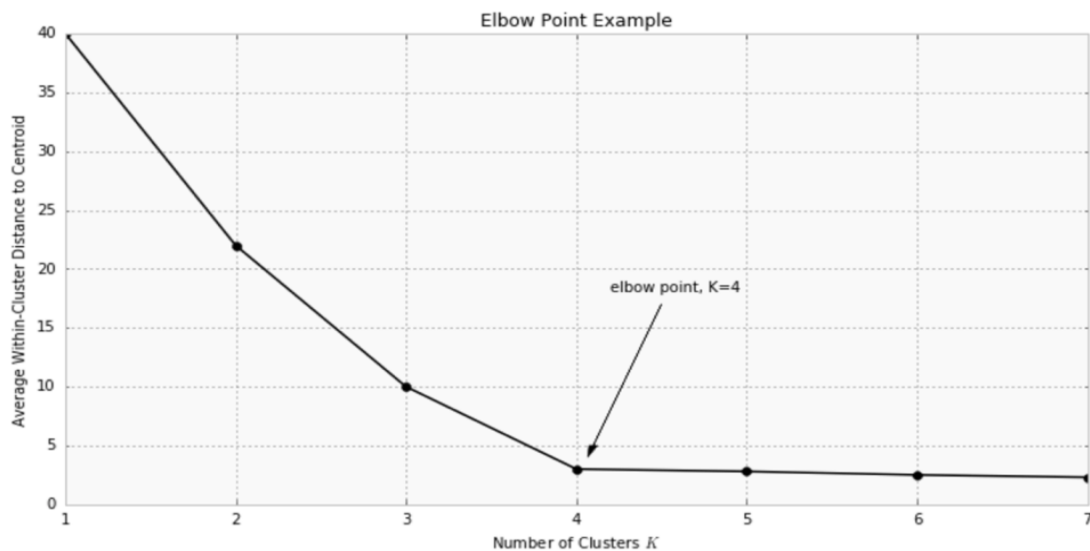


Figura 57. Ejemplo de función con punto de codo.

Es uno de los algoritmos más empleados a la hora de clasificar datos no etiquetados, ya que puede ayudar a encontrar relaciones entre los conjuntos de datos que a simple vista igual no podemos apreciar.

En nuestro caso, utilizaremos el algoritmo para identificar las posibles “clases” de caras que hay en los conjuntos de rostros de cada usuario. Nos interesa hallar el conjunto de la clase en la que estarán incluidas todas las caras del usuario.

### Implementación del algoritmo

Para la implementación práctica del algoritmo K-means nos beneficiaremos de la clase *Kmeans* que ofrece la librería *sklearn*. El proceso es sencillo, ya que lo único que debemos hacer una vez instanciada la clase será definir el número de clústeres o centros que deseamos obtener, y entrenar el modelo con los datos. La clase también nos permite la opción de predecir a cuál de los centros pertenece cada ejemplo, mediante el parámetro *labels*.

Dicho esto, solo nos quedaría recorrer la base de datos y aplicar el proceso citado por cada usuario, cribando los rostros y desechando las caras que no sean del usuario. La forma para realizar esta criba es la siguiente:

- **Primera iteración:** Inicialmente, y como resultado de haber aplicado el procesado de imágenes a través de la red neuronal, obtenemos los *embeddings* de un usuario almacenados en un diccionario.  
A pesar de que el recortado de caras de *OpenCV* es efectivo, se generan muchos recortes que no son caras. Necesitamos realizar una primera instancia del algoritmo K-means con únicamente dos clústeres, para diferenciar los rostros de los recortes fallidos. Por suerte, el número de caras correctas es notoriamente mayor, por eso, una vez generado el modelo y hecha la clasificación con todo el diccionario de *embeddings*, borramos los archivos de la base de datos y del diccionario de *embeddings*. Para realizar este borrado de recortes innecesarios, en primer lugar y mediante la función “mode” (moda, hace referencia a la función estadística, que devuelve la clase mayoritaria de una lista) obtenemos la clase mayoritaria, y en segundo lugar, aplicamos un filtrado de estas imágenes, quedándonos con solo aquellas que han sido predichas a la clase mayoritaria.
- **Segunda iteración:** Ahora disponemos de únicamente rostros, y la clasificación que viene a continuación resultará más efectiva. Este segundo proceso se realizará del mismo modo que el anterior. Generamos otra instancia de la clase *Kmeans* entrenándola con los *embeddings* del diccionario de rostros (ya habiendo quitado los recortes de rostros fallidos). Aprovechando la suposición que hemos defendido de que en cada usuario hay más fotos de su cara que del resto, estableceremos como rostros del usuario la clase mayoritaria tras aplicar el K-means por segunda vez. Al igual que hacíamos en la iteración anterior, desecharemos las demás caras. El número de clústeres empleado esta vez será de cuatro (después de varias pruebas, en nuestra base de datos es la configuración que más caras del usuario mantiene en el proceso de criba, de forma



generalizada). El borrado de las imágenes no necesarias se hace con la misma técnica que en la iteración anterior.

En la Figura 58 mostramos un ejemplo de ejecución en forma de esquema, de cómo se realiza el proceso de criba anterior, resaltando las imágenes que se eliminan en cada iteración.

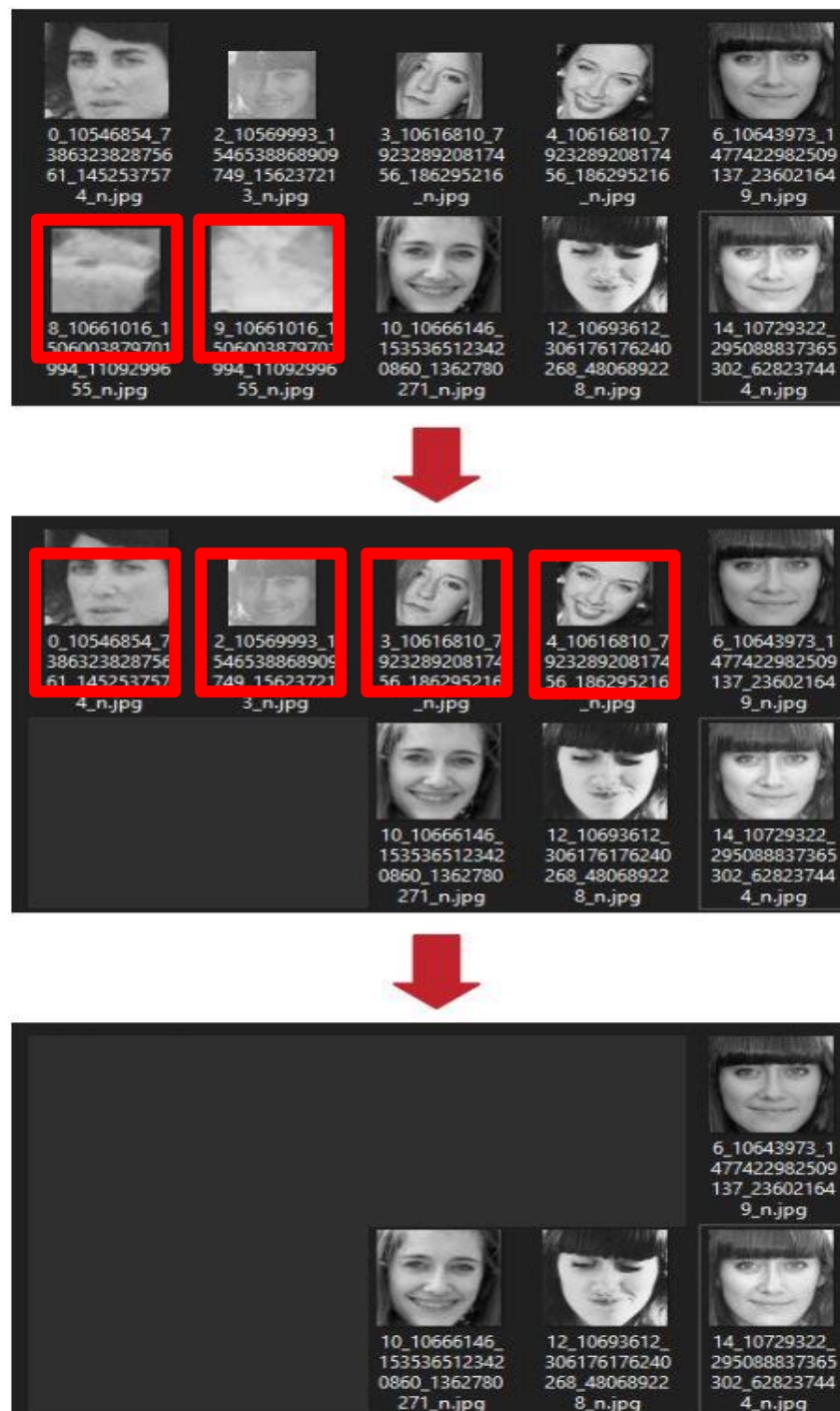


Figura 58. Iteraciones de la criba con K-means.

#### 4.2.4 Aumentando el conjunto de datos de entrenamiento (Data Augmentation)

Una vez hayamos identificado la cara del usuario, y hayamos descartado todas las demás, obtenemos un conjunto reducido de archivos que usaremos para entrenar próximamente nuestro sistema de reconocimiento facial.

En muchos usuarios, y sobre todo en las cuentas donde el número de fotos en las que aparece el rostro del usuario a identificar sea escaso, hace falta aumentar el tamaño del conjunto de entrenamiento para aumentar la efectividad del modelo. Pero, ¿cómo realizamos un aumento de datos solamente a partir de lo que ya disponemos? Efectivamente, no necesitaremos inventarnos nada nuevo, crearemos nuevos datos, lógicamente coherentes, siguiendo diferentes técnicas que hemos hecho mención anteriormente.

En el ámbito de la visión artificial, en donde es necesaria una gran cantidad de datos de entrenamiento, muchas veces como en nuestro caso es imposible recoger información nueva a parte de la que ya disponemos, que sería lo óptimo. En su defecto, se recurre a crear información, alterando de distintas formas la que ya tenemos.

Más concretamente si nos centramos en el campo del reconocimiento facial, además de las que ya hemos citado antes, existen muchas otras técnicas más específicas para aumentar nuestro conjunto de entrenamiento de rostros. Se distinguen tres tipos:

- *Transformaciones genéricas:* Estas se diferencian en geométricas (transformaciones espaciales de las caras, como rotaciones, giros ...) y fotométricas (cambios de color de la imagen, de contraste...).



Figura 59. Transformaciones geométricas. Fuente: [arxiv](#)



Figura 60. Transformaciones fotométricas. Fuente: [arxiv](#)

- *Transformación de componentes:* Alteramos componentes secundarios de la persona como puede ser el pelo, accesorios (gorros, gafas, ...) y apariencia en general.

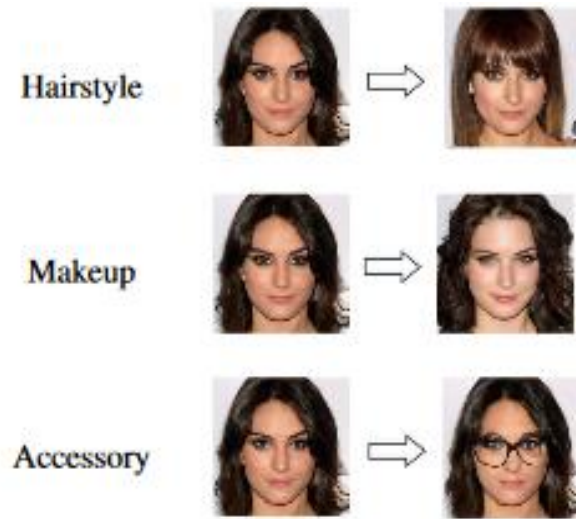


Figura 61. Ejemplos de transformación de componentes. Fuente: [arxiv](#)

- *Transformación de atributos:* Cambiando la pose, edad, la expresión del rostro y demás atributos temporales de la persona.

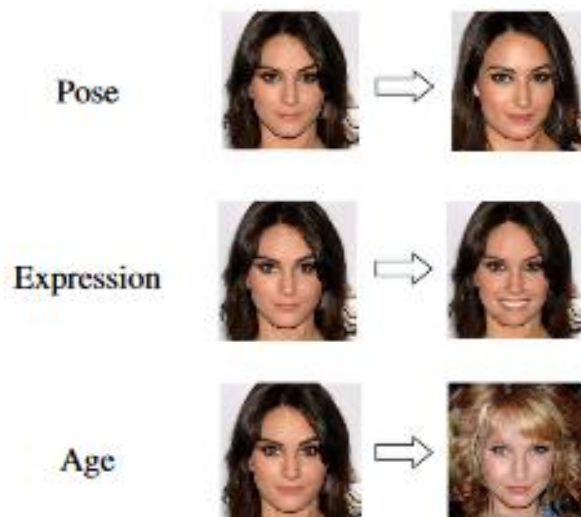


Figura 62. Ejemplos de transformación de atributos. Fuente: [arxiv](#)

Algunas de las transformaciones como por ejemplo la colocación de gafas o el cambio gestual requieren un gran coste computacional, ya que estos son el resultado de aplicar filtros de detección de puntos (en el caso de las gafas los ojos), o algoritmos de visión más costosos.

Para nuestro proceso descartaremos estas opciones y nos centraremos en las utilidades que nos ofrece la librería [imgaug](#). Esta nos permite, de forma eficaz realizar transformaciones sencillas a las imágenes para aumentar el tamaño del conjunto de fotos de entrada, como, por ejemplo, aplicar filtros *gaussianos* para meter ruido, rotar las imágenes, cambios de contraste, cambios de perspectiva... En total, la librería ofrece sesenta posibles transformaciones, que combinadas pueden dar lugar a más. En la siguiente imagen de la Figura 63 podemos observar el resultado de aplicar algunas de ellas:

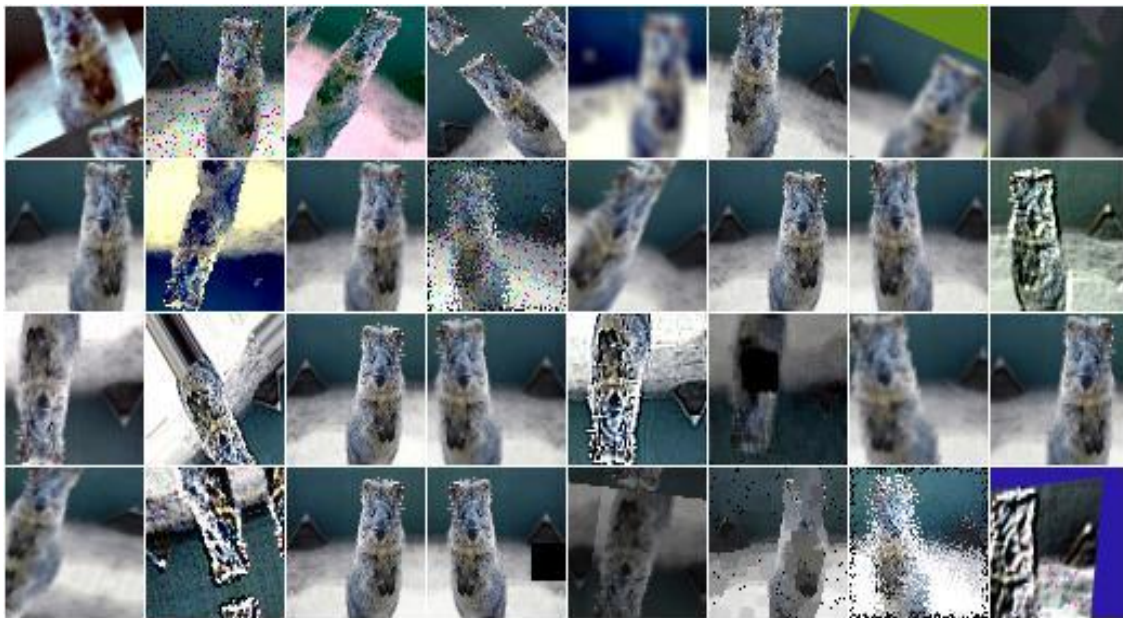
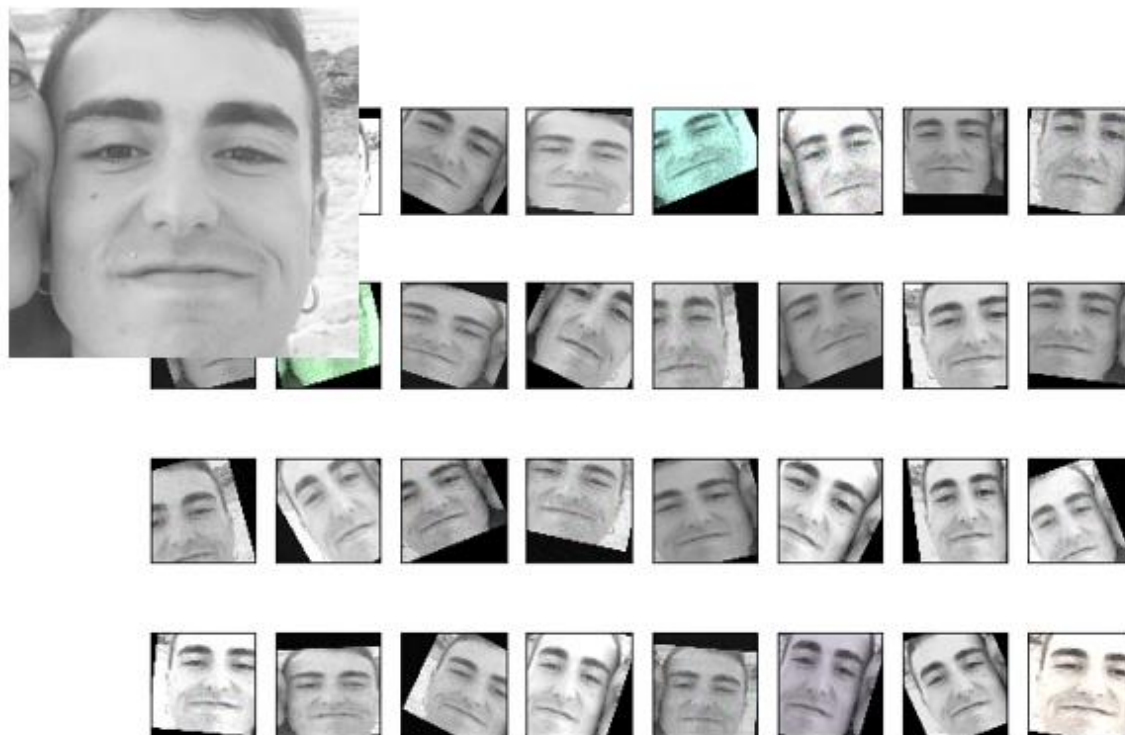


Figura 63. Transformaciones mediante *imgaug*. Fuente: [imgaug](#)

*Imgaug* trabaja a partir de unas secuencias probabilísticas, es decir, partiendo de unos parámetros que siguen cierta distribución se aplican transformaciones aleatorias. Nosotros, emplearemos una secuencia de parámetros predefinidos en la documentación de la librería, que por cada foto se realizan 32 transformaciones simples de la imagen.



A continuación, mostramos las imágenes resultantes tras aplicar la secuencia de transformación a un rostro:



*Figura 64. Transformaciones realizadas para el aumento de datos.*

De este modo, en el programa principal recorreremos una a una las caras de nuestra base de datos, aplicando la secuencia y generando nuevas imágenes a partir de estas. Finalmente, obtendremos 32 nuevas imágenes por cada cara del usuario que exista en la base de datos, aumentando el tamaño de datos significativamente. Si por ejemplo inicialmente disponemos de 20 rostros de un mismo usuario, tras aplicar este aumento de datos propuesto, nuestro conjunto de entrenamiento para ese usuario será de 640 rostros

## 5-Extracción de la información adicional: Perfilando a los usuarios

Después de haber obtenido todos los datos, en esta etapa del proyecto vamos a trabajar con ellos hasta extraer toda la información posible.

En nuestro caso analizaremos todas las fotografías de cada usuario, tanto las que aparecen personas como las que no, extrayendo de ellas posibles patrones que se repitan en el usuario, como pueden ser aficiones, gustos, tendencias... También es posible extraer información de los comentarios, saber el estado sentimental del usuario en cuestión comparando la tonalidad o significado de las palabras de los *posts* con un modelo textual, pero eso lo dejaremos de lado para centrarnos en el significado e información que podemos encontrar en las fotografías.

El objetivo principal de este apartado es conseguir averiguar los gustos del usuario, o aficiones, por medio de transformar las imágenes a texto y encontrar las palabras más relevantes de cada persona, para después poder conocerlas inmediatamente cuando el rostro del usuario sea reconocido. Con esto, conseguimos que si una persona a la que le agrada el motociclismo (averiguado gracias a que ha subido multitud de imágenes de motociclismo a Instagram y que el modelo de texto nos devuelve “motociclismo” como palabra relevante), entra a un bar y nuestro sistema de detección de rostro lo detecta, automáticamente la televisión podría ofrecerle una carrera de motor para agradar al usuario la estancia.

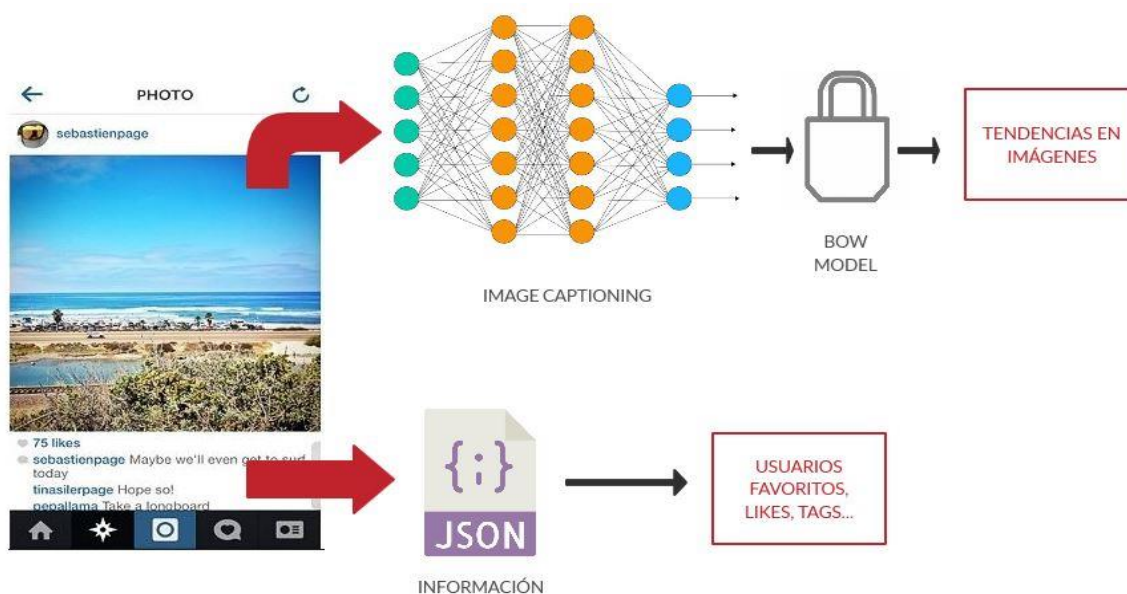


Figura 65. Esquema general de la extracción de información adicional.

En la anterior Figura 65, y al igual que hicimos con el enfoque general del proceso, mostramos un esquema visual de la extracción de la información adicional. De esta forma podemos ver como por un lado extraemos de las publicaciones información directa en un fichero *JSON* como puede ser el promedio de *likes*, usuarios más frecuentados en los comentarios... Y por otro lado que extraemos de las imágenes que previamente hemos recolectado con el *scraper*.

## 5.1 *Image captioning*

El *Image Captioning* (o *Caption Generator*), es un problema conocido de inteligencia artificial que se basa en a partir de una imagen, dar una descripción simple a modo de pie de foto de lo que se puede ver en la imagen (*caption*).

El problema para un ser humano es sencillo, ya que para una mente humana no nos cuesta trabajo relacionar una foto con objetos dispuestos en un plano del espacio. Pero para un ordenador, es un problema de aprendizaje profundo ya que es necesario reconocer los objetos o seres que aparecen en la foto, junto con la acción que estén realizando, disposición espacial ...

Para abordar el problema, y como veremos a continuación, haremos uso de las redes neuronales.

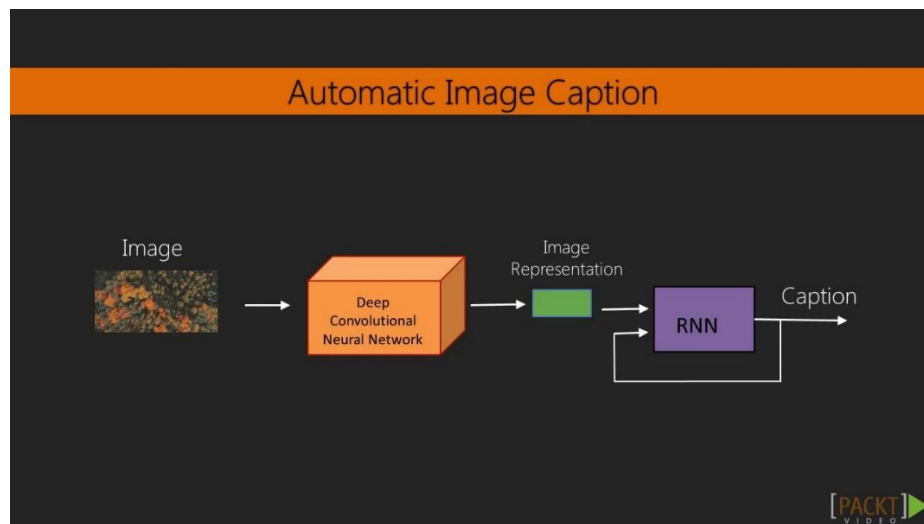


Figura 66. Esquema general de los modelos de red neuronal. Fuente: [Packt](#)

En la Figura 66 podemos observar el funcionamiento general y la disposición del modelo de red empleado que describiremos a continuación.



Inicialmente disponemos de una imagen que será la entrada a la red, y la salida será la descripción de esta. El modelo general viene a ser el resultado de concatenar dos arquitecturas de red distintas a lo largo de todo el proceso de transformación. Primero se procesa la imagen de entrada a través de una red convolucional (CNN) obteniendo una representación vectorial de la misma (*embedding*), y posteriormente esta se procesa a través de una red recurrente (RNN). El número de llamadas a la misma red recurrente de la propia red vendrá dado por la longitud de descripción de salida deseada, ya que, en cada llamada, se predecirá una palabra, siendo la suma de todas ellas el resultado final o *caption* final que devolverá el modelo como salida.

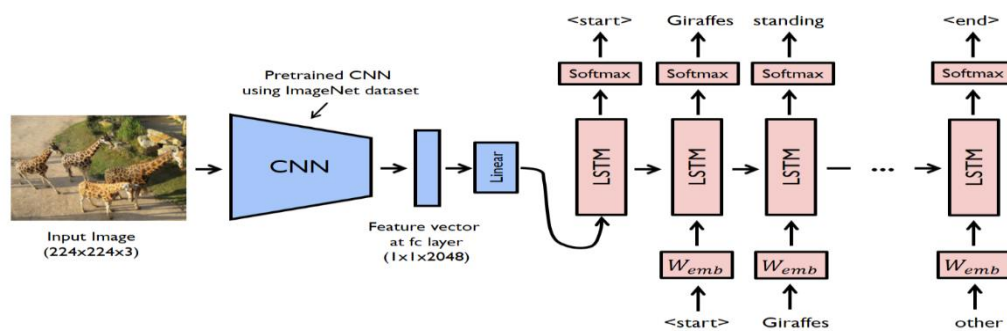


Figura 67. Llamadas recurrentes realizadas al modelo. Fuente: [yunjeu](#)

Mediante la red convolucional, obtendremos un *embedding* de dimensión  $1 \times 1 \times 2048$  resultado de la aplicación final de capas *fully connect* (FC). El conjunto de entrenamiento de esta red necesariamente tiene que ser grande si queremos obtener un mínimo de precisión en los resultados, ya que se trata de un problema aparentemente sencillo, pero con un margen de error bastante grande. Para este tipo de problemas existen variedad de *datasets* con imágenes libres de copyright, las más típicas son:

- **COCO dataset:** Se trata de un conjunto de imágenes de 83000 archivos. Es el más empleado para este tipo de problemas ya que abarca la mayoría de los tipos distintos de imágenes. Al ser un conjunto muy grande de entrenamiento y no disponer de los medios suficientes para entrenar una red de tal magnitud, hemos tenido que descartar esta opción, aun siendo la mejor, porque nos llevaría mucho tiempo.
- **Flickr 16K:** Es un conjunto de imágenes de libre acceso originarias de la famosa página Flickr. Está compuesta por un total de 16000 archivos. A pesar de tener un tamaño más reducido, resulta eficaz para imágenes de ámbito social como las que pueden ser de una red social ya que estas pueden coincidir notablemente con las que podemos encontrar en Flickr.

- **Flickr 8K:** Igual que el anterior solo que de tamaño más reducido todavía, 8000 imágenes. Este ha sido el *dataset* empleado para nuestra elaboración, ya que a pesar de no obtener mucha precisión en los resultados nos sirve para experimentar y entender cómo funciona el modelo y aprender a solucionarlo.

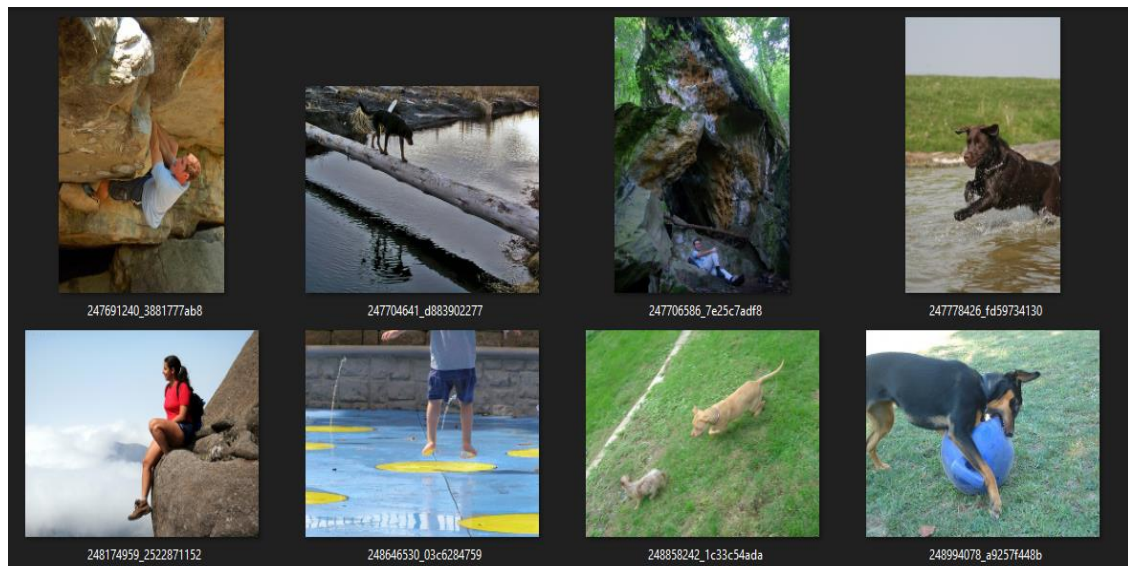


Figura 68. Captura de la disposición del dataset Flickr8K.

Todos los *datasets* por defecto tienen tres subconjuntos, *train*, validación y *test*, y todos vienen acompañados de ficheros de texto, donde se almacenan las descripciones correctas de las imágenes correspondientes. En el caso del *dataset* de Flickr8K, el conjunto de *train* se compone de 6000 imágenes, 1000 de prueba y 1000 de validación. Los ficheros de descripciones enlazan el nombre de la imagen en cuestión con la descripción de esta.

247691240_3881777ab8.jpg#4	There is a man in a light shirt climbing a rocky cliff .
247704641_d883902277.jpg#0	A black dog is slowly crossing a fallen log that is outstretched over a stream of water .
247704641_d883902277.jpg#1	A dog crossing a river on a bridge made of a fallen tree .
247704641_d883902277.jpg#2	A dog walks on a log across a small river .
247704641_d883902277.jpg#3	The black dog is walking along a tree trunk bridge over water .
247704641_d883902277.jpg#4	The dog walks across the stream on a fallen log .
247706586_7e25c7adf8.jpg#0	A man in a blue shirt and jeans poses by large mossy rocks .
247706586_7e25c7adf8.jpg#1	A man is laying under a large mossy rock in the forest .

Figura 69. Estructura de los ficheros que contienen las descripciones.

### 5.1.1 Implementación del modelo

Para entender cómo es el funcionamiento completo del proceso de *Image Captioning*, hemos realizado una implementación sencilla del modelo basándonos en las opciones que ofrece Keras y mediante el *dataset* Flickr8K antes citado.

El objetivo de este apartado es únicamente entender el funcionamiento del modelo, ya que para conseguir un error muy bajo hace falta un proceso de entrenamiento muy profundo y tener los recursos suficientes. A pesar de haber utilizado un entrenamiento progresivo como vamos a ver, y a pesar de utilizar Google Colab, este no permite la ejecución de código más de doce horas seguidas, por lo que no podemos entrenar el modelo al completo. Al no disponer de esos recursos, para generar los resultados, hemos utilizado otro modelo completamente distinto previamente entrenado.

A continuación, analizaremos paso a paso cómo hemos realizado la implementación de la aplicación:

### Dataset

Como ya hemos mencionado antes, emplearemos el *dataset* de Flickr8k, ya que se trata de un *dataset* relativamente pequeño (8000 imágenes), gratuito, con el que podemos entrenar el modelo y tenemos unos resultados aceptables empleando una CPU básica, además resulta ser bastante realista.

Una vez descargado el *dataset* obtenemos dos archivos:

- Flickr8k\_Dataset.zip un archivo con todas las fotografías.
- Flickr8k\_text.zip Un archivo con todas las descripciones de las fotografías.

### Preparación de dataset de fotos

Usaremos un modelo pre-entrenado para interpretar el contenido de las fotos. Existen multitud de modelos que elegir, en este caso, emplearemos el Oxford Visual Geometry Group, o VGG. Se trata de una red convolucional típica que se usa para este tipo de problemas, mediante la cual realizamos la primera parte del modelo general, crear los *embeddings*. Este modelo recibirá una fotografía y devolverá el *embedding* de tamaño 1x4096 correspondiente.

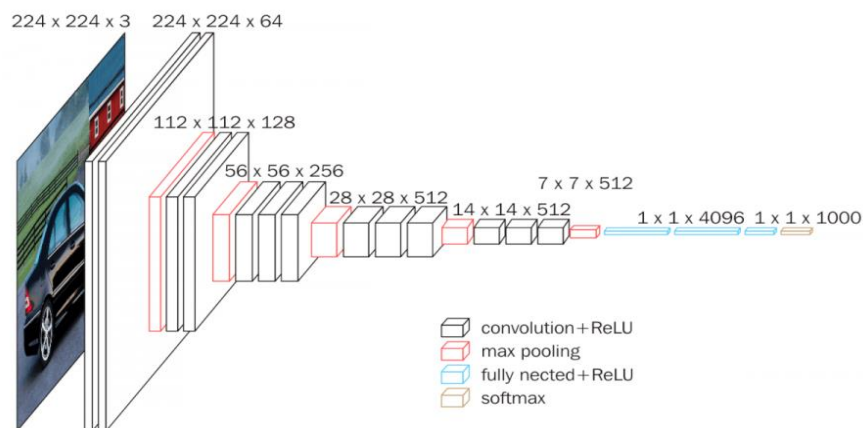


Figura 70. Estructura de capas del modelo VGG16. Fuente: [neurohive.io](http://neurohive.io)

Keras proporciona el modelo pre-entrenado directamente, descargándolo la primera vez que se use a través de internet (500Mb). El problema es que, al ser un modelo bastante grande, el procesar cada foto por el modelo cada vez que queramos testearla será redundante, y en su lugar, haremos un precómputo de las fotos total y guardaremos los resultados del primer procesado, es decir, los *embeddings*, en un archivo, así no tenemos que hacerlo cada vez que queramos ejecutar la red (optimización para consumir menos memoria).

Una vez cargado el modelo, y si ejecutamos el comando *summary*, podemos ver con detalle las capas que lo componen. Si nos fijamos, la última capa del modelo nos devuelve un vector de salida de tamaño 1000, correspondiente al número de clases total. Esta no nos interesa por lo que necesitamos reestructurar el modelo haciendo un *pop* de la última capa, para quedarnos con la predicción que realiza la penúltima.

=====		
input_1 (InputLayer)	(None, 224, 224, 3)	0
block1_conv1 (Conv2D)	(None, 224, 224, 64)	1792
block1_conv2 (Conv2D)	(None, 224, 224, 64)	36928
block1_pool (MaxPooling2D)	(None, 112, 112, 64)	0
block2_conv1 (Conv2D)	(None, 112, 112, 128)	73856
block2_conv2 (Conv2D)	(None, 112, 112, 128)	147584
block2_pool (MaxPooling2D)	(None, 56, 56, 128)	0
block3_conv1 (Conv2D)	(None, 56, 56, 256)	295168
block3_conv2 (Conv2D)	(None, 56, 56, 256)	590080
block3_conv3 (Conv2D)	(None, 56, 56, 256)	590080
block3_pool (MaxPooling2D)	(None, 28, 28, 256)	0
block4_conv1 (Conv2D)	(None, 28, 28, 512)	1180160
block4_conv2 (Conv2D)	(None, 28, 28, 512)	2359808
block4_conv3 (Conv2D)	(None, 28, 28, 512)	2359808
block4_pool (MaxPooling2D)	(None, 14, 14, 512)	0
block5_conv1 (Conv2D)	(None, 14, 14, 512)	2359808
block5_conv2 (Conv2D)	(None, 14, 14, 512)	2359808
block5_conv3 (Conv2D)	(None, 14, 14, 512)	2359808
block5_pool (MaxPooling2D)	(None, 7, 7, 512)	0
flatten (Flatten)	(None, 25088)	0
fc1 (Dense)	(None, 4096)	102764544
fc2 (Dense)	(None, 4096)	16781312
predictions (Dense)	(None, 1000)	4097000
=====		
Total params: 138,357,544		
Trainable params: 138,357,544		
Non-trainable params: 0		

Figura 71. Summay del modelo VGG16.

Keras proporciona herramientas para hacer un *reshape* de la foto a una 3x224x224, más eficiente para la red.

A continuación tenemos la función *extract\_features()*, que lo que hace es, dado el nombre de una carpeta, carga cada foto, la prepara para pasársela a la red VGG mediante el *reshape*, y recoge las características devueltas por el modelo reconvirtiendo cada imagen en un vector 1x4096 (aplicando el proceso de generación de *embeddings* que hemos explicado antes). La función devuelve un diccionario de los *ids* de las fotos y sus vectores. Una vez llamada la función obtenemos un archivo llamado *features.pkl* donde se almacena el resultado de los *embeddings* generados. La ejecución de esta etapa en una CPU normal tarda aproximadamente una hora.

### Preparación del dataset de descripciones

El *dataset* que vamos a trabajar a continuación contiene las descripciones de las imágenes de entrenamiento. Pero antes de empezar a trabajar con el texto, debemos hacer un pre-procesado y limpieza de texto.

Cada foto tiene un *id* único que será empleado también con las descripciones. A partir del fichero donde se encuentran las descripciones, ejecutaremos una función que lee el contenido del archivo y devuelve un diccionario referenciando las *ids* de las fotos con sus descripciones.

Posteriormente y como hemos explicado antes, necesitamos hacer una limpieza de texto para que sea más fácil de utilizar y trabajar. En este caso haremos: convertir todas las letras en minúsculas, quitar signos de puntuación, quitar las palabras de longitud 1 y quitar las palabras que contengan números.

```
Vocabulary Size: 8918
Out[6]:
```

	word	count
0	a	62989
1	.	36581
2	in	18975

Figura 72. Ejemplos de aparición.

Aquí podemos observar (Figura 72) la frecuencia de aparición de algunas palabras antes de realizar el proceso de limpieza. Obviamente, el signo de puntuación como puede ser el punto nos resulta irrelevante, y tras la limpieza, la carga computacional será mucho menor.



Una vez limpio, ya podemos obtener nuestro propio vocabulario, el cual nos interesa que sea lo más reducido posible y finalmente guardar el diccionario resultante en un fichero.

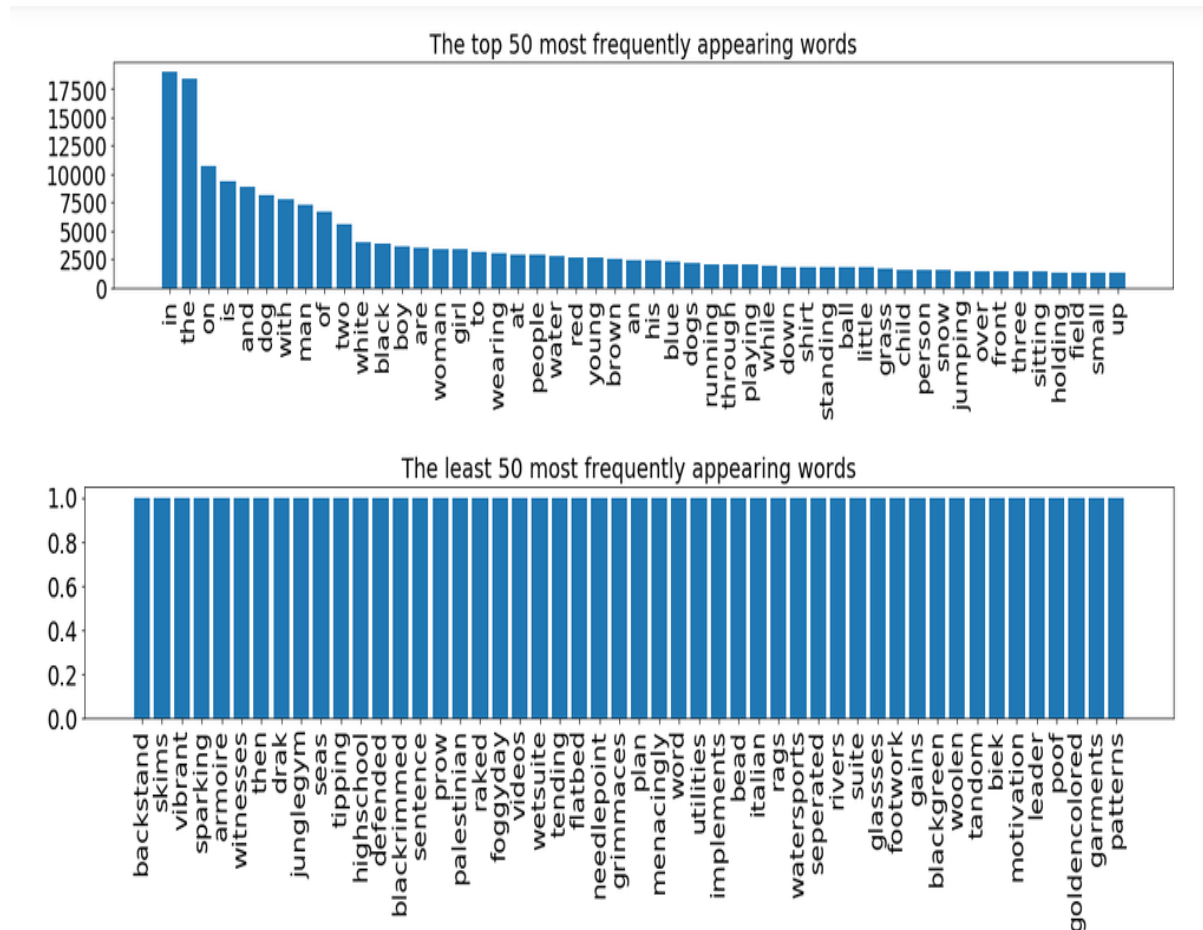


Figura 73. Palabras más y menos frecuentes de nuestro vocabulario.

Finalmente, para terminar de manipular el texto, “tokenizamos” el vocabulario y las descripciones. Es decir, convertimos las palabras en números indexados, de forma que las descripciones sean vectores de números, lo que resulta más sencillo a la hora de entrenar el modelo y de paso ahorramos memoria de almacenamiento y ejecución. El “tokenizado” se almacena en un fichero, con los índices las palabras, el cual utilizaremos a modo de diccionario cada vez que necesitemos generar una descripción nueva.

### Imágenes similares

Del mismo modo que ya habíamos realizado en pasos anteriores, aplicaremos el algoritmo PCA para ver fotografías similares y visualizarlas de forma más distinguible. Si plasmamos todas las imágenes del *dataset* en un mismo gráfico obtenemos:

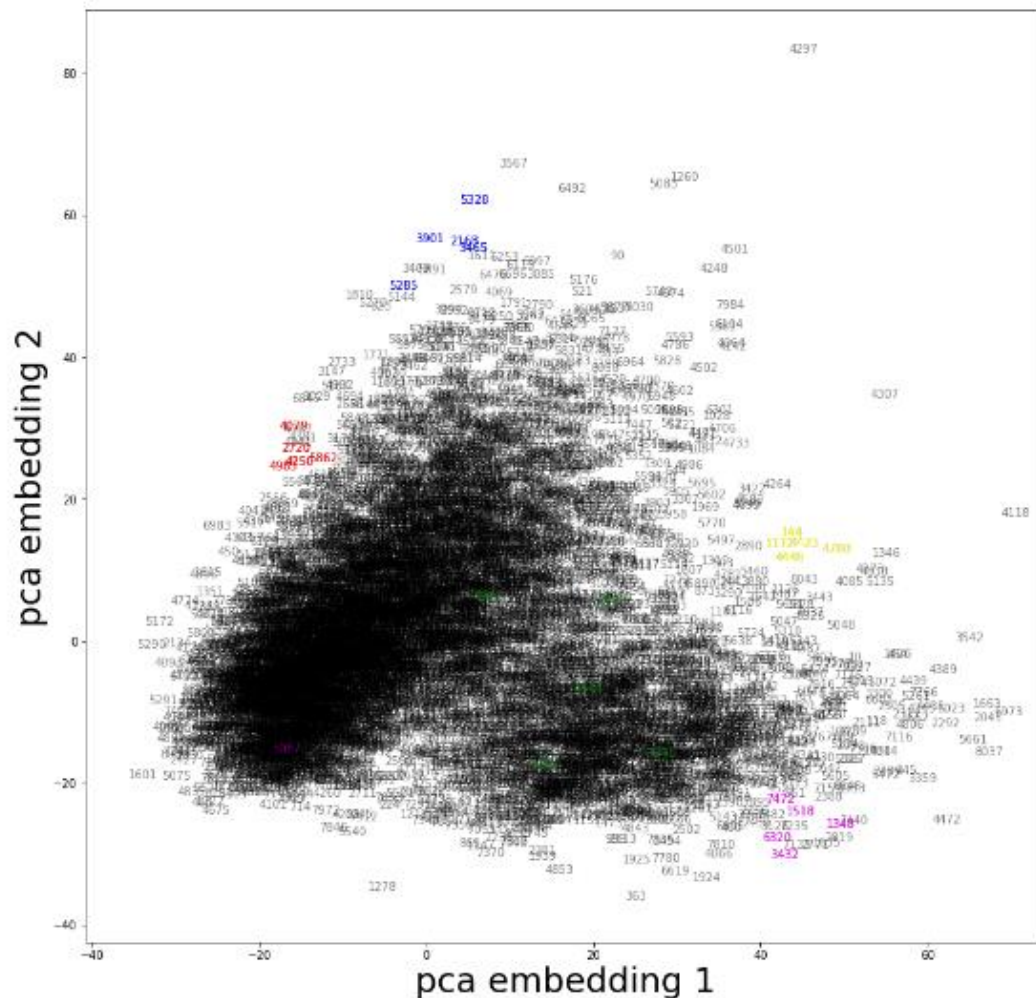


Figura 74. Visualización de imágenes parecidas entre sí.

Cada punto hace referencia al nombre de la foto del *dataset*. Hemos coloreado el nombre de algunas, con el mismo color las que más cerca están entre sí, para mostrarlas a continuación y comprobar si realmente adquieren cierto parecido visualmente también.



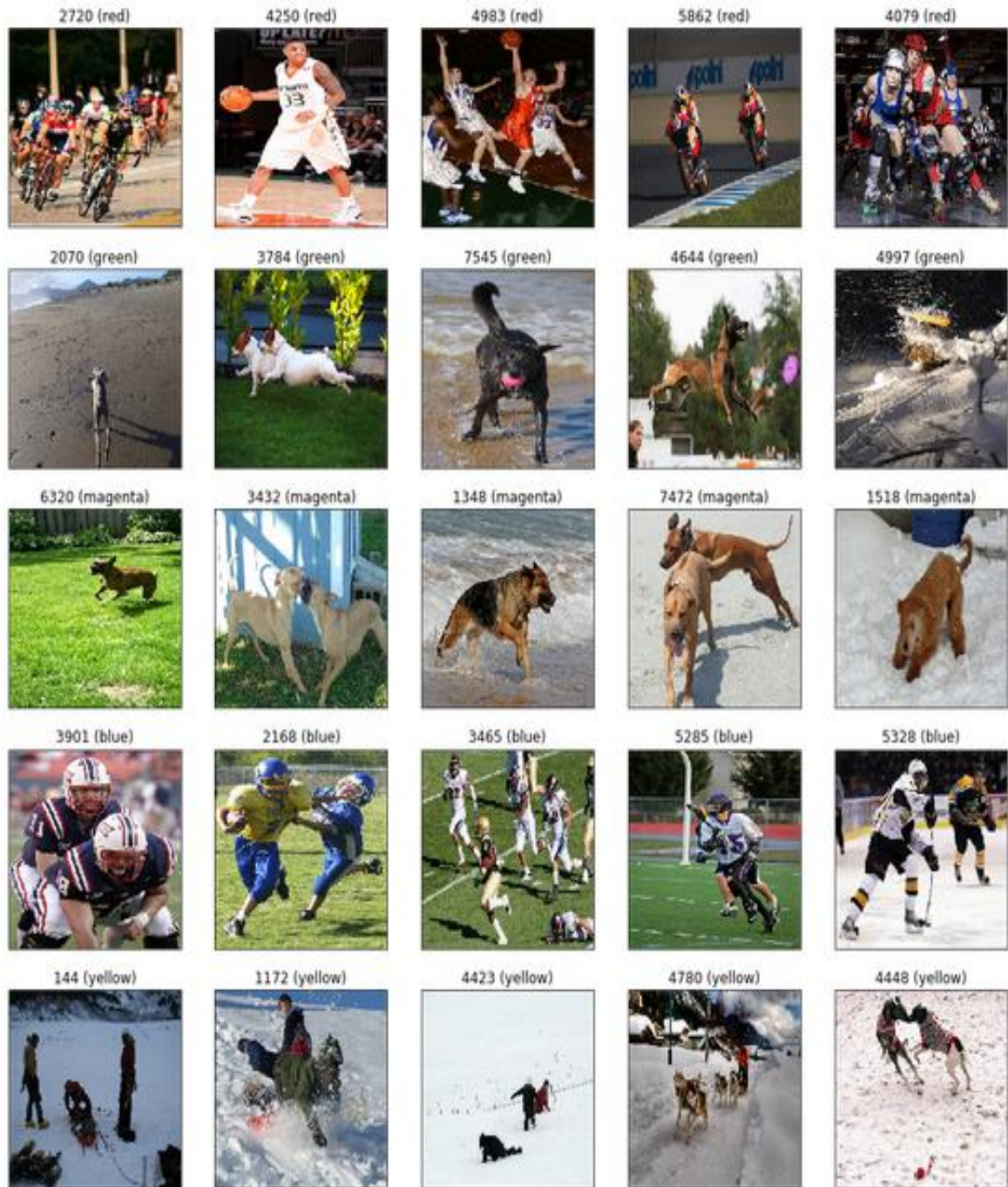


Figura 75. Imágenes próximas.

Como podemos observar en los ejemplos mostrados de cada “clase”, las fotos que están más próximas en el gráfico según sus dos nuevas características comparten patrones o rasgos.

### Modelo LSTM

Una vez recorrido el primer modelo y generado los *embeddings*, y procesado y hecho la limpieza de las descripciones, definimos la red neuronal recurrente que será la que genere las descripciones de las futuras imágenes.

A este modelo se le realizarán tantas llamadas recurrentes como palabras queramos que se generen. Una vez más, emplearemos la librería LSTM que proporciona Keras para definir el modelo secuencial.

Una vez creado, ya podemos enlazar los dos modelos anteriores en una sola estructura:

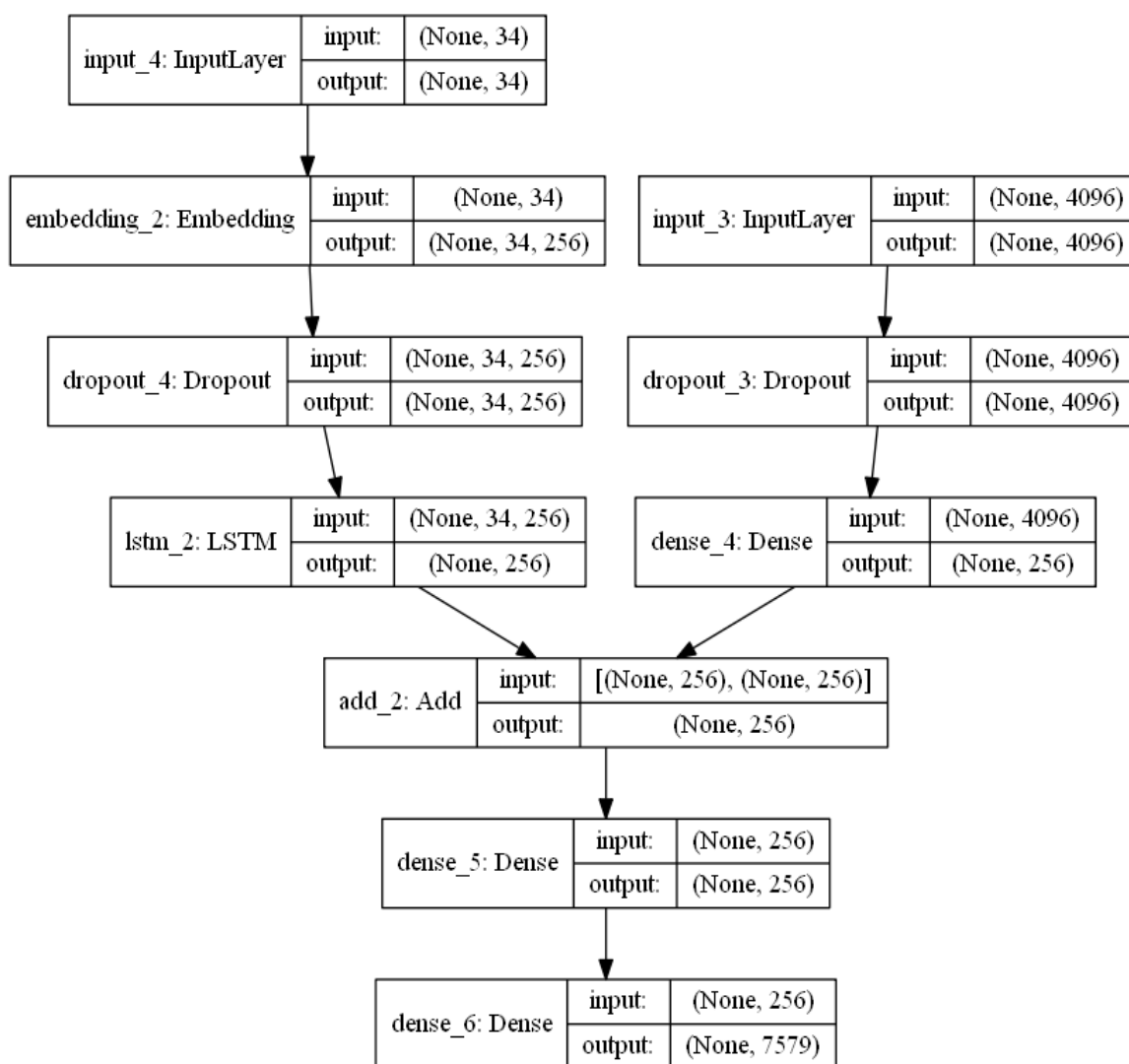


Figura 76. Estructura del modelo general en detalle.

### Entrenamiento del modelo

Al no disponer de recursos para realizar un entrenamiento completo de la red, hemos realizado un cargado progresivo de datos, que consiste en realizar un entrenamiento por lotes. Con esto nos ahorramos gran cantidad de uso de la CPU (o GPU en el caso de usar Colab), ya que de la otra manera nos daba fallo de memoria al no disponer de la suficiente.

Realizamos una función de entrenamiento en donde la red recibe el fichero de *embeddings* generado en pasos anteriores, y el fichero con nuestro vocabulario y descripciones. La función en cada iteración de entrenamiento realiza una vuelta entera (también llamada época) al conjunto de datos, modificando los pesos. Una vez finalizada cada una de estas iteraciones, el programa almacena automáticamente en un fichero de nombre *model\_num\_epoca.h5* el estado del modelo junto a sus pesos (la ejecución de cada época tarda alrededor de una hora).

En nuestro caso, a pesar de intentar ejecutar el algoritmo para que realice 20 épocas, lo máximo que hemos conseguido ejecutar ha sido 11, con un valor en la función de coste de 3.37 (*loss*).

Una vez terminado el proceso, bastará con cargar el contenido del fichero en una variable para evitar tener que entrenar el modelo cada vez que lo vayamos a usar.

En la siguiente gráfica de la Figura 77 podemos observar la variación de la función de coste en las 5 primeras épocas de ejecución:

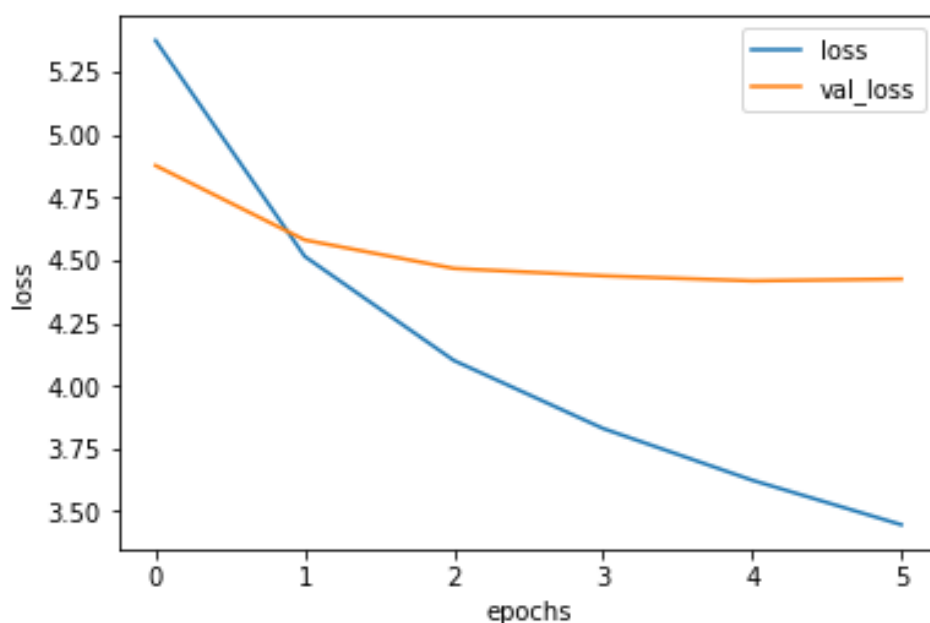


Figura 77. Variación de la función de coste.

### Generación de nuevas descripciones

A partir de este punto, hemos adaptado el algoritmo a la resolución de nuestro problema particular. Principalmente, la idea es recorrer la carpeta donde se almacenan las imágenes de un usuario generando por cada foto la descripción equivalente, e ir almacenando las descripciones en un archivo.

Dicho de otro modo, por cada imagen que haya en la carpeta, crearemos el *embedding* resultante del modelo VGG y los iremos almacenando en un fichero llamado *usuario.pkl*.

Una vez obtenidos todos los *embeddings* del usuario, el programa recibirá los *embeddings* y el fichero que contiene el vocabulario “tokenizado” para dar salida a las correspondientes descripciones en el fichero *desc\_usuario.txt*. Más adelante veremos qué hacemos con este fichero de texto (Figura 78).

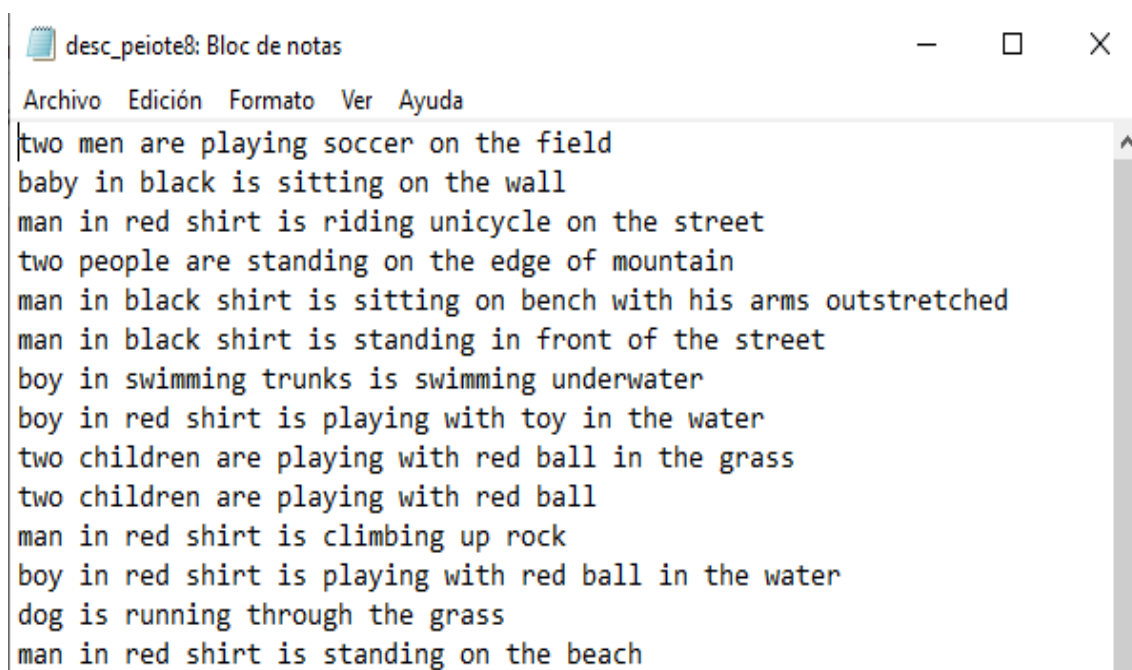


Figura 78. Ejemplo del formato del archivo de descripciones resultante.

### Resultados y pruebas

Como ya se ha mencionado antes, al tratarse de un *dataset* pequeño, y además no tener los recursos suficientes para el entrenamiento completo de la red, los resultados son bastante deficientes en comparación con la precisión requerida.



Con las fotos sacadas de Instagram, los resultados son peores todavía, ya que depende del tipo de fotos del usuario, la claridad de estas... Pero, a pesar de obtener un valor elevado en la función de coste, en algunos casos el algoritmo predice correctamente la descripción de la foto.



Figura 79. Ejemplo de descripción bien predicha.

**Predicción:**

*boy in swimming trunks is swimming underwater*



Figura 80. Ejemplo de una mala predicción.

**Predicción:**

*man in red shirt is riding unicycle on the Street*

A pesar de que, como hemos mencionado antes, las predicciones que realiza nuestro modelo no son muy buenas, otros modelos más entrenados con *datasets* más grandes en dimensión de ejemplos, en comparativa no son mucho mejores que este, por lo que he decidido emplear el que he implementado yo.

## 5.2 Tendencias del usuario

En este apartado, explicaremos el proceso de cómo a partir de los ficheros de texto con las descripciones generados, conseguimos identificar patrones o tendencias del usuario.

La idea principal de esta parte es analizar todas las palabras que aparecen en los ficheros de descriptores y llegar a identificar aquellas palabras que nos aporten cierta información del usuario.

### 5.2.1 *Modelo Bag of Words*

El modelo *Bag of Words* (Bolsa de Palabras, o simplemente, *BoW*), es un problema de aprendizaje automático orientado a textos, en donde se miden las frecuencias de aparición de las palabras en los distintos tipos de textos, para después realizar un clasificado de estos (por ejemplo, para clasificar si un artículo pertenece a uno de ámbito científico o a un cuento). Este modelo se basa en transformar las frases en vectores según sus palabras (si nos fijamos esto tiene cierta similitud con el “tokenizado” que hicimos en el *image captioning*).

Nosotros hemos cogido la idea principal que tiene la implementación del modelo *BoW* para realizar un programa que muestre las palabras más relevantes de un usuario.

### 5.2.2 *Palabras relevantes en las descripciones*

El modelo *BoW* lo que hace es generar un vocabulario propio para un corpus (colección de textos o documentos) dado, midiendo todas las apariciones de las palabras. Nosotros, crearemos un modelo por cada usuario.

Nos ayudaremos de la clase *CountVectorizer* que ofrece la librería *sklearn* para crear el vocabulario y “tokenizar” el texto.

#### Implementación

El primer paso que debemos hacer es leer el fichero donde se encuentran las descripciones y almacenarlo en una variable como si fuese un *string*. Aquí, si dispusiéramos de más documentos, cada texto es almacenado en un *string* y finalmente serían enlazados mediante una lista. A esta variable la llamaremos “corpus”.

Ahora, ya podemos iniciar el objeto *CountVectorizer* llamando al constructor:

```
# Se importa la librería CountVectorizer
from sklearn.feature_extraction.text import CountVectorizer
# Se llama al constructor de la clase CountVectorizer
vectorizer = CountVectorizer(stop_words = 'english') |
```

Figura 81. Invocación del objeto *CountVectorizer*.

De esta forma almacenamos en la variable *vectorizer* el modelo que formará el texto con las descripciones. El argumento de *stop\_words* se refiere a que, a la hora de entrenar el modelo, eliminaremos las palabras irrelevantes o que no aportan información para el modelo (en este caso en inglés), como pueden ser “in”, “the”, “are” ...

Una vez instanciado, el modelo ya puede ser entrenado con el corpus de nuestro usuario:

```
# Se realiza el aprendizaje a partir del corpus y se transforma que los valores aprendidos
corpusTransformado = vectorizer.fit_transform(corpus)
# Se muestran los términos aprendidos (variables del modelo BoW)
palabrasAprendidas = vectorizer.get_feature_names()
print(palabrasAprendidas)

['arm', 'arms', 'baby', 'background', 'ball', 'beach', 'bench', 'black', 'boy', 'boys', 'camera', 'children', 'climbing',
'dog', 'edge', 'field', 'girl', 'girls', 'grass', 'hand', 'head', 'holding', 'man', 'men', 'mountain', 'outstretched', 'p
eople', 'phone', 'playing', 'red', 'riding', 'rock', 'running', 'shirt', 'sidewalk', 'sitting', 'soccer', 'standing', 'st
reet', 'swimming', 'toy', 'trunks', 'underwater', 'unicycle', 'wall', 'water', 'young']
```

Figura 82. Entrenado del modelo BoW.

En la Figura 82 podemos observar cómo llamamos a la función de entrenamiento del objeto y las palabras que han sido aprendidas.

Si accedemos al parámetro *vocabulary.items()* de la variable *vectorizer*, podemos ver el diccionario que se ha creado con las palabras aprendidas junto al número de apariciones que podemos encontrar cada palabra a lo largo del texto:

```
dict_items([('men', 23), ('playing', 28), ('soccer', 36), ('field', 15), ('baby', 2), ('black', 7), ('sitting', 35), ('wa
ll', 44), ('man', 22), ('red', 29), ('shirt', 33), ('riding', 30), ('unicycle', 43), ('street', 38), ('people', 26), ('st
anding', 37), ('edge', 14), ('mountain', 24), ('bench', 6), ('arms', 1), ('outstretched', 25), ('boy', 8), ('swimming', 3
9), ('trunks', 41), ('underwater', 42), ('toy', 40), ('water', 45), ('children', 11), ('ball', 4), ('grass', 18), ('climb
ing', 12), ('rock', 31), ('dog', 13), ('running', 32), ('beach', 5), ('girl', 16), ('girls', 17), ('arm', 0), ('backgroun
d', 3), ('young', 46), ('boys', 9), ('phone', 27), ('camera', 10), ('sidewalk', 34), ('holding', 21), ('head', 20), ('han
d', 19)])
```

Figura 83. Diccionario de palabras y frecuencias.



Y si de este diccionario devolvemos las  $n$  palabras (en este caso 15) con más apariciones a lo largo del texto, estaríamos devolviendo las 15 palabras que más se repiten en las descripciones de las fotografías.

```
palabras = dev_n_usadas(vectorizer.vocabulary_, 15, 0)
print(palabras)

['young', 'water', 'wall', 'unicycle', 'underwater', 'trunks', 'toy', 'swimming', 'street', 'standing', 'soccer', 'sitting', 'sidewalk', 'shirt', 'running']
```

Figura 84. Palabras más repetidas de un usuario.

Una vez obtenidas las palabras más frecuentes del usuario, ahora tenemos que quedarnos únicamente con las importantes o con las que realmente nos interesan. Para ello, hemos descargado un fichero de texto el cual contiene alrededor de 2000 aficiones o hobbies. Lo leemos e introducimos todas las aficiones de ejemplo en una lista llamada “hobbies” (obtenida de [proyecto](#)).

```
print(hobbies[0:25])

['3d printing', 'abseiling', 'acting', 'action figure', 'adventure racing', 'aerobatics', 'aeromodeling', 'aggressive inline skating', 'aid climbing', 'aikido', 'air hockey', 'air racing', 'air sports', 'airbrushing', 'aircraft spotting', 'airsoft', 'airsofting', 'aizkolaritza', 'alpine skiing', 'amateur astronomy', 'amateur radio', 'amateur astronomy', 'amateur geology', 'amateur pankration', 'amateur radio']
```

Figura 85. Fichero de hobbies.

El último paso que tenemos que hacer, es comprobar si alguna de las palabras con mayores frecuencias está incluida en la variable hobbies:

```
relevantes = []
for pal in palabras:
    if pal in hobbies:
        relevantes.append(pal)
print(relevantes)

['underwater', 'swimming', 'soccer', 'running']
```

Figura 86. Tendencias finales del usuario.

Con esto concluimos el proceso de extracción de información de los datos, obteniendo las coincidencias del usuario, para después, una vez reconocido al usuario, podamos devolver a modo de resultado. Estas son almacenadas en el fichero de relevancia.

## 5.3 Extracción de información adicional del usuario

Además de extraer información por medio de predicciones, como hemos conseguido a través del *Image Captioning*, podemos recopilar información directa que da el usuario, como pueden ser comentarios o *tags*.

Aplicando la opción de *-comment* en el *scrapeo* de los datos, el autómata además de recoger las imágenes recoge más información almacenándola en un fichero *json*, a modo de base de datos. De estos datos todavía podemos extraer más información del usuario para poder mostrarla una vez reconocido el rostro.

Por lo tanto, en esta etapa del proyecto tratamos de recorrer todos los usuarios, analizando los ficheros *json* de cada uno de ellos y extrayendo a su vez toda la información relevante posible. Python ofrece una extensa librería en cuanto a los ficheros *json*, por lo que hace sencilla su utilización.

A continuación, explicamos cuál es la información que hemos catalogado como provechosa, de la cual pensamos que puede resultar interesante y mostramos el proceso que se ha seguido para conseguir extraer cada una de ellas (cada información de cada usuario ha sido almacenada en un fichero distinto, siguiendo la lógica de la estructura de la base de datos que hemos detallado en la introducción del capítulo).

### 5.3.1 Popularidad

Hemos creado una función que clasifica al usuario según unos criterios que hemos diseñado (según intereses se puede cambiar los límites), nos clasifica al usuario dentro de un rango A-E de "popularidad" siendo A *muy popular* y E *poco popular*.

Esta clasificación se realiza a partir de los datos de número de publicaciones del usuario y de los *likes* que tiene cada publicación (y de ahí la media de *likes* por publicación), extraídos del fichero *json*.

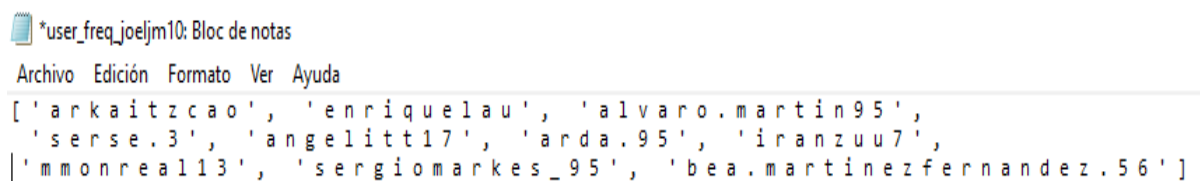
Una vez tengamos los resultados, se almacenan en el fichero *prom\_likes\_usuario.txt*, adoptando la siguiente forma:

```
prom_likes_carlosalvarezm97:
Archivo  Edición  Formato  Ve
3 2 . 7 3 3|
E
```

Figura 87. Popularidad.

### 5.3.2 Usuarios frecuentes

Hemos extraído todos los nombres de usuario distintos al usuario que estamos analizando, contabilizando el número de veces que aparece a lo largo de todos los comentarios. De forma similar a como realizábamos el modelo *BoW*, en esta parte también ordenamos las palabras (en este caso los nombres de usuario) según las frecuencias de aparición, y nos quedamos con los *N* que más se repiten. De este modo, obtenemos en el fichero *user\_freq\_usuario.txt* un listado de las personas con las que más interactúa nuestro individuo:



\*user\_freq\_joeljm10: Bloc de notas

Archivo Edición Formato Ver Ayuda

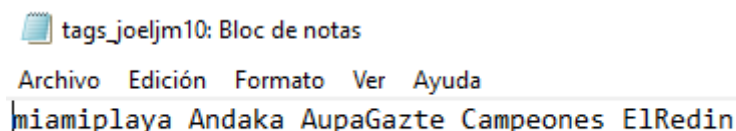
```
[ 'arkaitzca0', 'enriquela0', 'alvaro.martin95',
  'serse.3', 'angelitt17', 'arda.95', 'iranzuu7',
  'mmonreal13', 'sergiomarkes_95', 'bea.martinezfernandez.56' ]
```

Figura 88. Usuarios frecuentes.

### 5.3.3 Tags o tendencias reales

A partir de los títulos de las publicaciones también podemos obtener los *tags* reales que el usuario introduce manualmente. A diferencia de las predicciones realizadas por el *Image Captioning*, estos *tags* son seguros ya que son tendencias reales del usuario.

Al igual que con los usuarios frecuentes, realizamos un modelo con los *tags* que aparecen en todas las fotos midiendo la frecuencia de aparición de cada una, y almacenando en el fichero *tags\_usuario.txt* únicamente las *N* *tags* más relevantes en orden de aparición:



tags\_joeljm10: Bloc de notas

Archivo Edición Formato Ver Ayuda

```
miamiplaya Andaka AupaGazte Campeonos ElRedin
```

Figura 89. Tags.

### 5.3.4 Emojis más usados

Empleando el mismo procedimiento que en los casos anteriores, según los comentarios realizados por el usuario analizado guardamos en *emoji\_usuario.txt* los *N* emoticonos favoritos del usuario. Para el tratamiento de emoticonos, Python ofrece la librería *Emoji*, que facilita su uso extrayendo directamente los *emojis* de los *strings*.

Esto, realmente, no creo que pueda aportar gran información a diferencia del resto de las cosas extraídas, pero puede resultar interesante conocer su uso. A continuación, mostramos cómo quedan almacenados en los ficheros y cómo quedan en el programa:

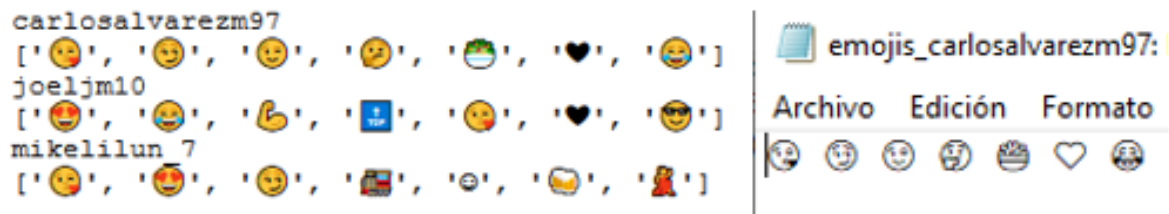


Figura 90. Emojis.

### 5.3.5 Títulos de las publicaciones

Almacenamos también los títulos de las publicaciones en el archivo *títulos\_fotos\_usuario.txt*. Para tener más datos a la hora de predecir tendencias con el *Image Captioning*, hemos introducido los títulos de las publicaciones dentro del fichero de descripciones generado antes, y así luego ejecutar otra vez el modelo *BoW*.

Antes de incluir los títulos, se ha limpiado el texto y se ha hecho uso de la librería *goslate*, para traducir el texto a inglés (las descripciones predichas están en inglés). Esta librería, sorprendentemente muestra una gran precisión en las traducciones, ya que realiza correctamente incluso las que originalmente están en euskera.

### 5.3.6 Comentarios

Tanto los comentarios realizados por el usuario, como los comentarios de otros usuarios han sido previamente pre-procesados, eliminando caracteres extraños, emoticonos, tendencias y nombres de usuario, y almacenados en los ficheros correspondientes *comentUser\_usuario.txt* y *comentNoUser\_usuario.txt*.

Mas adelante, en el apartado de líneas futuras analizaremos por qué esta información, sobre todo los comentarios realizados por el usuario, resultarán de un alto grado de interés.

## 6-Reconocimiento facial

Una vez generados todos los datos de interés en los distintos ficheros, y obtenido los *embeddings* de las caras correspondientes a cada usuario, hemos llegado a la última parte del proceso.

En este apartado final, se procederá a recoger imágenes de video, reconocer a los individuos y mostrar toda la información en pantalla.

### 6.1 Reconocimiento a tiempo real

Al igual que hicimos para detectar y recortar las caras de las imágenes, haremos uso de *OpenCV*. El procedimiento es sencillamente el mismo, pero en vez de recortar una cara de una imagen física, el recorte lo realizamos sobre una grabación de video a tiempo real. Esto se consigue gracias a la función *VideoCapture(p)*, que crea un objeto del cual podemos tomar fotogramas a tiempo real con la función *read*, de una cámara conectada en el puerto *p*. En nuestro caso, hemos empleado la *web-cam* integrada en el portátil alojada en el puerto 0, pero si queremos emplear una externa bastará con indicar el puerto correspondiente. También hemos hecho pruebas con una cámara deportiva conectada como dispositivo externo para recoger a través de ahí las imágenes, situando esta junto a la entrada de una puerta simulando la entrada de personas (podremos ver la demostración en el video). El obturador de las cámaras deportivas es de tipo ojo de pez, deformando un poco las caras (coge mayor ángulo de visión), por lo que los resultados no son tan eficientes como los obtenidos con la *web-cam*.

El algoritmo de obtención de imágenes es sencillo: se trata de un bucle infinito donde estamos constantemente leyendo fotogramas de forma dinámica. A la vez, en la ventana generada como interfaz, dibujamos un marco sobre el rostro detectado con *OpenCV*, con los datos equivalentes a la persona reconocida. A continuación, explicaremos cómo se realiza la clasificación de reconocer a la persona.

## 6.2 Clasificación

El proceso de clasificación es sencillo, ya que al tener los *embeddings* de la base de datos previamente almacenados en un diccionario, lo único que tenemos que hacer es, por cada fotograma generado en el bucle, crear el *embedding* con el modelo y compararlo con los demás.

Para conocer cuál es el grupo de *embeddings* más parecidos al de entrada, hemos creado distintas funciones con la finalidad de recorrer el diccionario aplicando distintos criterios de comparación (en todos ellos se calculará por cada usuario la distancia Euclídea de la foto de entrada con respecto a todas las fotos de cada uno y asignará una etiqueta de salida según estos criterios):

- Mínimo: Nos quedamos con la persona que devuelva la distancia mínima de todas las distancias. Es decir, con la persona que tenga la cara almacenada más parecida (según la distancia Euclídea) a la de entrada.
- Media: Se calculará una media de las distancias de las caras de cada usuario respecto a la de entrada, y se asignará la predicción a la persona con menor distancia media.
- N-media: Basándose en el concepto anterior, pero esta vez solo tendremos en cuenta las N distancias más pequeñas de cada usuario para calcular la media de distancias.

Como veremos en el apartado de gráficas, el criterio de quedarnos con el mínimo con bases de datos pequeñas da muy buenos resultados, pero a medida que el tamaño de nuestra base de datos aumenta, este resulta ser poco preciso. El criterio de N-media en nuestra implementación no es apreciable la diferencia, pero para este tipo de problemas de reconocimiento es la que mejores resultados muestra.

A su vez, también comentaremos que se ha introducido el término de umbral en nuestra implementación. De tal manera que, si el valor devuelto por estas funciones es demasiado elevado, significará que la imagen de entrada no es parecida a ninguna persona presente en la base de datos, por lo que prediciremos a esa persona como "Desconocida".

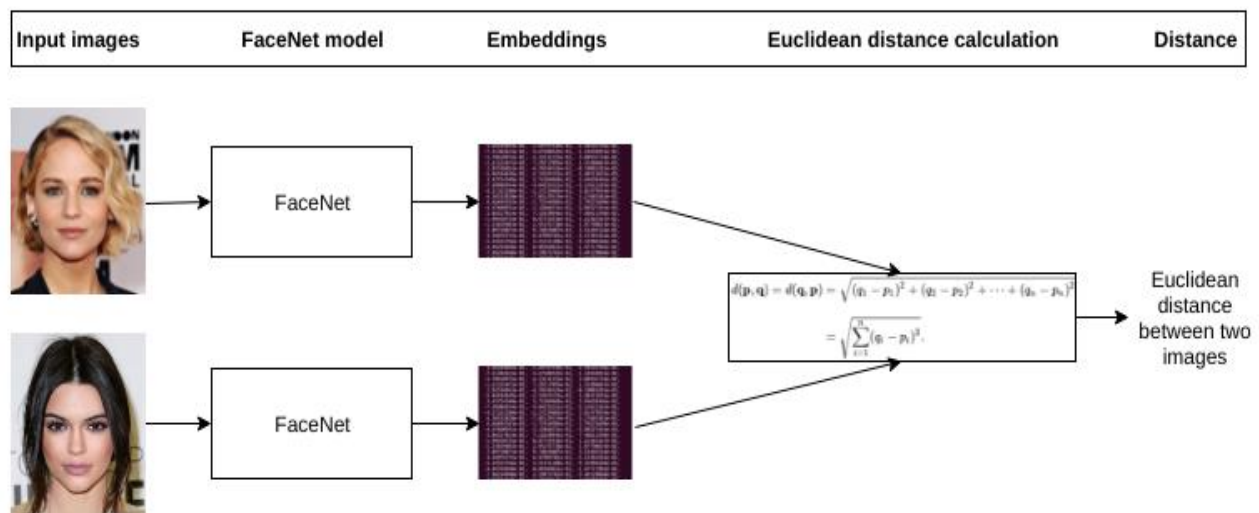


Figura 91. Proceso de comparación de dos rostros mediante FaceNet. Fuente: [mc](#)

### 6.3 Resultados y gráficas de rendimiento

En este apartado mostraremos un desglose de los resultados de eficacia obtenidos, así como algún ejemplo de ejecución del programa. Cuando hablemos de eficacia a lo largo de este apartado, nos centraremos en la eficacia mostrada al realizar el reconocimiento de la persona y no en la detección de rostro (de *OpenCV*), ya que ha sido el proceso más trabajado por nosotros. Para realizar un análisis de eficacia completo, tendríamos que emplear alguna técnica de búsqueda para hallar el ajuste de parámetros óptimo ya que la eficacia en el caso de nuestra implementación depende de varias variables (criterio de clasificación, umbral, tamaño de la base de datos, número de predicciones, ...) y otros factores ajenos a la codificación (luminosidad del entorno, obturador empleado, factores impredecibles, ...). Nosotros, ya que hemos observado cuáles son los factores que más alteran la precisión de acierto del algoritmo, vamos a realizar una comparativa de cómo varía el resultado según el tamaño de la base de datos (número de usuarios introducidos en la base de datos) y el criterio de clasificación empleado (media, N-media o mínimo).

Para recopilar los datos de ejecución, hemos ejecutado el programa una vez con cada ajuste, de forma que se realice el reconocimiento de 500 fotogramas en cada ejecución. El porcentaje de aciertos (o *accuracy*), lo calcularemos como el resultado de la relación entre los fotogramas que han sido bien clasificados y los que no. Es decir, mediremos la precisión a la hora de reconocer a una persona (decir a que usuario de la base de datos hace referencia el rostro detectado) y trataremos como acierto si el nombre de usuario coincide con el que realmente es, y fallo si realiza otro tipo de predicción.



Para poner esto en práctica, lo que he hecho ha sido incluir mi usuario en la base de datos en todas las ejecuciones y ponerme yo delante de la cámara. Entonces, *accuracy* se calculará como:

$$100 * \frac{n^{\circ} \text{ de veces que pred} = \text{mi usuario}}{\text{total de fotogramas clasificados}}$$

Considero que esta medición no es la óptima, ya que como ya he dicho antes, intervienen más factores (por ejemplo, hay veces que el *accuracy* varía según las personas que haya en la base de datos, ya que a algunas las confunde más que a otras), pero esta medición puede servirnos como orientativa para hacernos una idea de cuán bueno es el algoritmo.

En la gráfica que tenemos a continuación (Figura 92), disponemos de un desglose de los resultados obtenidos en el análisis de eficacia general. En el eje Y, los porcentajes de acierto y, en el eje X, el número de personas que formaban la base de datos en esa ejecución. Cada barra vertical hace referencia al criterio de clasificación empleado, tal y como se muestra en la leyenda. En la tabla auxiliar de la parte inferior de la figura obtenemos un desglose del valor exacto de los datos obtenidos.

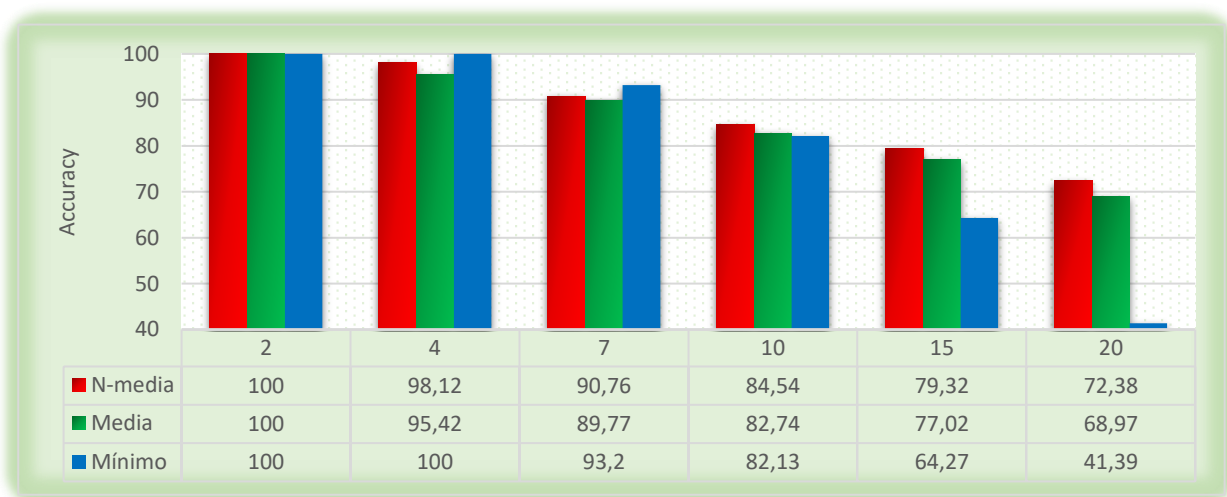


Figura 92. Gráfica con los resultados del análisis de eficacia.

Sorprendentemente, los resultados han sido mejor de lo esperados, ya que las cuentas con las que he realizado las pruebas no tienen una cantidad muy grande de rostros. Es decir, en las cuentas personales que nosotros hemos empleado para formar la base de datos no hay muchas imágenes que contengan caras de personas, por lo que el número de rostros por usuario almacenados es pequeño, lo que producirá que el margen de fallo será mayor. En Instagram es muy habitual publicar imágenes de paisajes u otros elementos donde no hay rostros, es por ello por lo que ese margen de fallo dependerá de los usuarios que formen la base de datos y de los contenidos que estos publiquen. En cuentas de gente famosa, por ejemplo, en donde el número de imágenes es mayor, habrá más caras detectadas, en consecuencia, el algoritmo dará mejores resultados.

En las siguientes fotografías que se muestran en las páginas que vienen a continuación, obtenemos tres ejemplos de ejecución, para tener una idea visual de cómo queda la interfaz con el usuario. En la Figura 93 se muestra una detección de mi propio usuario personal. En la Figura 94, otra detección sobre una fotografía impresa, esta vez, del perfil del cantante Pello Repáraz. En la Figura 95 se muestra una detección de una persona ajena a la base de datos junto al reconocimiento al mismo tiempo de mi usuario.

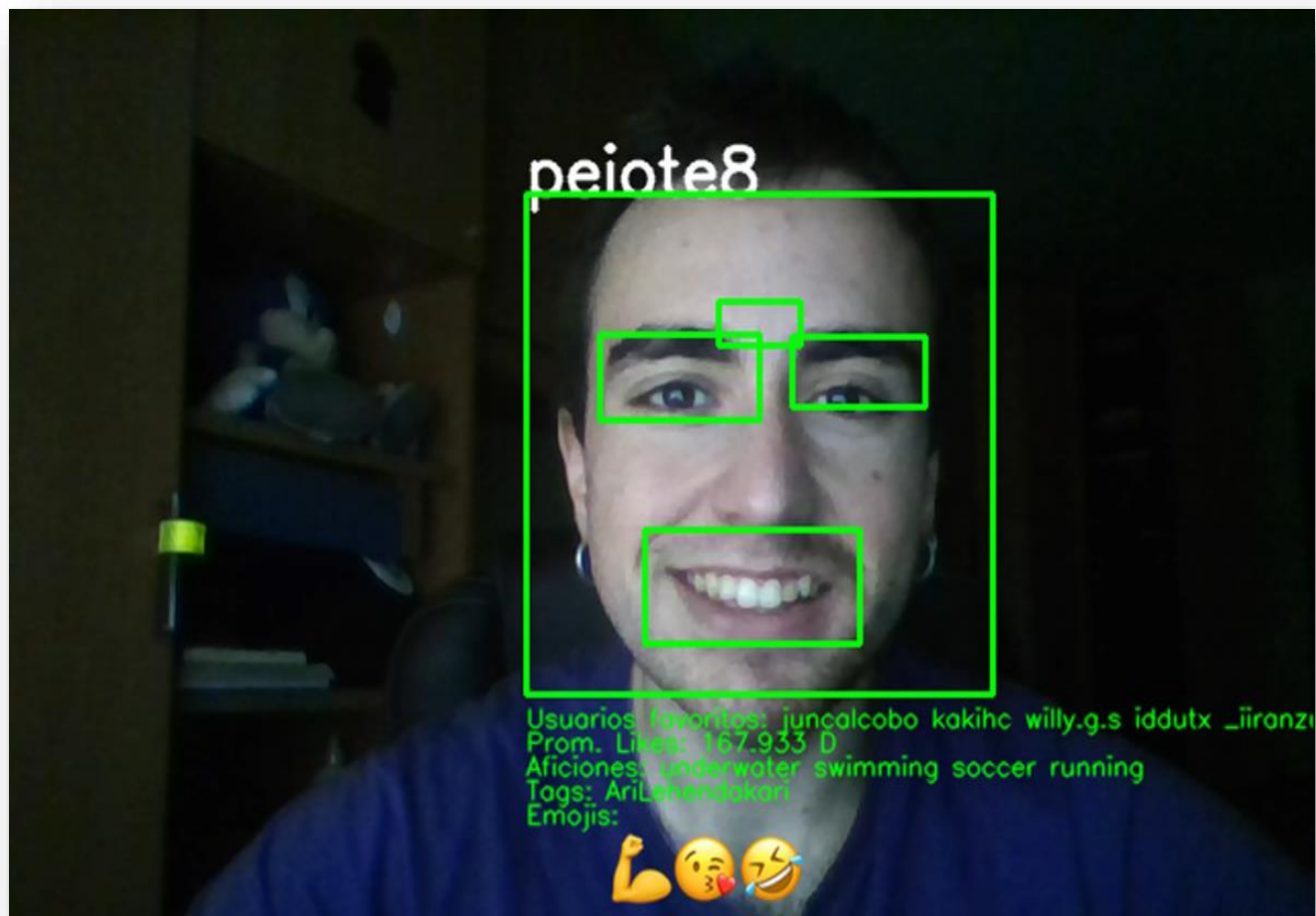


Figura 93. Ejemplo de ejecución 1.



Figura 94. Ejemplo de ejecución 2.

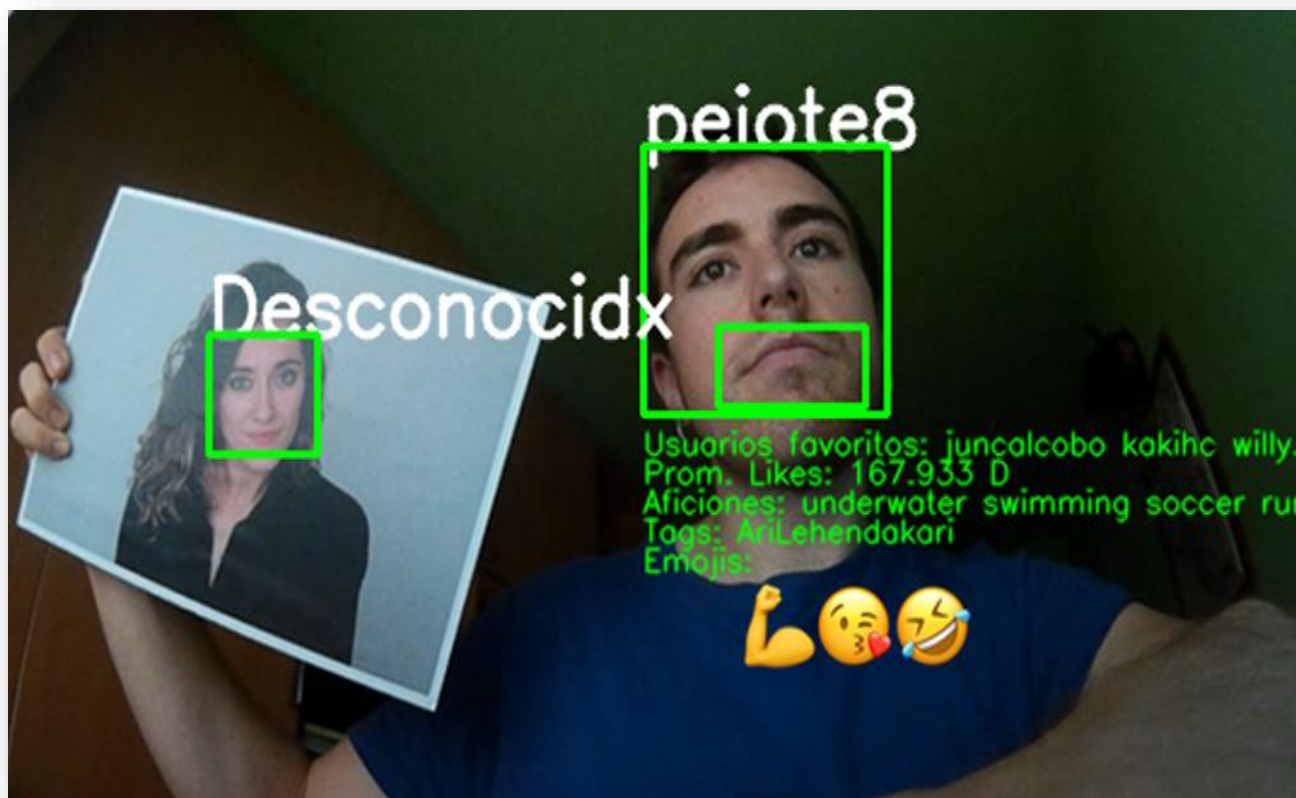


Figura 95. Ejemplo de ejecución 3.

Por otro lado, también hemos probado el reconocedor que implementa *OpenCV*, mediante *face.LBPHFaceRecognizer\_create()*. Esta implementación se basa en el algoritmo creado en el año 1996 de LBPH *Local Binary Patterns Histograms* (siendo una extensión del LBP del año 1994).

Este algoritmo se basa en el análisis de las imágenes según las texturas y la interpolación de los píxeles. Si añadimos el concepto de histogramas, nos permite representar las imágenes en vectores a través de las intensidades de los píxeles (consiguiendo representar la imagen en vectores, pero de tamaño muchísimo mayor al que conseguimos con los embeddings). El entrenamiento se realiza de forma parecida al nuestro, almacenando los histogramas en forma de vector para posteriormente realizar la comparativa (computacionalmente deberá hacer más comparativas). Esto no nos interesa ya que, al no emplear técnicas *Deep Learning*, la utilización de este método con grandes cantidades de datos y millones de fotos por usuario sería impensable. Otro factor negativo es que, al no ser un procedimiento automatizado, este método es vulnerable a modificaciones de los datos, y no aprende de los resultados como las técnicas que nosotros empleamos.

## 7-Conclusiones

Una vez finalizada la elaboración del proyecto en su totalidad, hemos abordado el objetivo principal de conseguir un sistema de reconocimiento facial a partir de datos extraídos de Instagram. El sistema resultante es completamente funcional y hemos podido comprobar que su precisión es relativamente buena aun disponiendo de pocos datos de entrenamiento y una base de datos reducida en cuanto al número de usuarios.

En conclusión, podemos decir que, disponiendo de recursos limitados y con las técnicas precisas, se puede realizar una extracción de información de los datos aparentemente buena. Como luego analizaremos en líneas futuras, la mejora del reconocimiento facial requiere de más esfuerzo y de una mejor adaptación de los recursos, pero su mejora sería completamente posible.

Es por ello por lo que debemos concienciarnos sobre todo el impacto que puede tener lo que estamos compartiendo a través de las redes sociales, ya que, si en un simple proyecto de fin de grado, una sola persona ha sido capaz de extraer toda esta información, no quiero ni imaginar lo que las grandes empresas son capaces de recolectar.

Si nos paramos a pensar en posibles aplicaciones que puede llegar a tener este sistema en la vida real, podemos poner hipotéticas situaciones en las que sería de utilidad:

- En un bar, detectar la tendencia musical de un usuario y cambiar automáticamente la música.
- En una tienda de ropa detectar que color de ropa utiliza más y lanzar ofertas en prendas de ese color.
- Publicidad personalizada según aficiones, en general.
- Detectar perfil de personalidad para saber cómo tratar con esa persona ante situaciones determinadas.
- ...



## 8-Líneas futuras

En este apartado se proponen y detallan ideas que se podrán aplicar para generar nuevas versiones mejoradas del proyecto.

### 8.1 Modelo *Big Five*

El modelo *Big Five*, o en castellano modelo de los cinco grandes, es un modelo psicológico que describe la personalidad de una persona a través de cinco factores: Neuroticismo, Agradable, Responsabilidad, Extraversión y Experiencia ([Modelo Big Five. Fuente: Wikipedia.](#)). En psicología, se realiza un estudio a través de cuestionarios basándose en los cinco rasgos que define el modelo *Big Five*, mediante el cual, según el grado de cada factor resultante, permite encajar a la persona con un tipo de personalidad u otra.

Este modelo ha despertado mucha curiosidad en las redes sociales (sobre todo en Twitter) ya que, mediante él, se puede llegar a conocer la personalidad de una persona analizando las palabras empleadas por el usuario en cuestión (cada palabra tiene unos porcentajes asociados a cada factor del modelo). Hasta ahora, se han realizado estudios para detectar [Cyberbullying](#) o incluso para analizar el perfil de personalidad de algunos famosos.

Si esta idea la trasladamos a nuestro proyecto, teniendo almacenados todos los comentarios que realiza el usuario, si analizamos todas las palabras empleadas en cada comentario rigiéndonos por este modelo, resultaría de mucho interés obtener el perfil de personalidad del usuario reconocido y mostrar los resultados a la vez que los demás datos.

Como este ejemplo de uso podríamos poner muchos, ya que, en psicología existen muchos modelos en donde pueden estar relacionados con datos personales extraídos masivamente. Gracias a estos, podemos inferir directamente en la mente del usuario jugando con los distintos modelos. Sin ir más lejos, un factor muy relacionado con el diseño y usabilidad en la informática es la Psicología Cromática (como afecta en el estado de ánimo percibir unos colores u otros), en donde se juega con unos colores u otros según lo que queramos transmitir al usuario. A donde quiero llegar con todo esto es a qué si empleamos los datos de forma adecuada, existen muchas opciones de extraer información de estos empleando distintas técnicas. Con el anterior ejemplo, una vez conozcamos la personalidad del usuario y basándonos en la Psicología Cromática,

podemos establecer una serie de reglas de tal manera que, si se detecta que una persona tiende a ser neurótico, nervioso, automáticamente emplearemos colores azules para transmitir tranquilidad. Esto es un ejemplo de uso entre muchos posibles.

## 8.2 Aplicación Móvil

Puede ser interesante trasladar esta aplicación a una *app* móvil para probar el funcionamiento también en otra plataforma completamente distinta, y comprobar su comportamiento si además la base de datos es almacenada en la nube.

Por ello, se propone en un futuro trasladar la implementación del proyecto a un *smartphone*.

## 8.3 Mejorar eficiencia del proceso

A lo largo de este proyecto, la mayoría de los modelos empleados han sido modelos pre-entrenados, y la base de datos de personas empleada ha sido reducida con la idea de emplear únicamente datos accesibles, sencillos. En cuestión, la mayoría de los datos y herramientas empleadas han sido enfocadas a los recursos que yo dispongo (ya que sería por ejemplo intratable entrenar una red neuronal o realizar una descarga de *scrapeo* de millones de datos para montar la base de datos desde un ordenador de uso personal). Es por eso por lo que, aunque se hayan logrado los objetivos del proyecto, muchos de los puntos tratados se podrían hacer de forma más eficiente y que diese mejores resultados en cuanto a eficiencia, como son los casos del *image captioning* o del reconocimiento facial. Si tenemos bases de datos más grandes, con más caras o más datos, las detecciones y predicciones también serán mejores. En un futuro y con los recursos adecuados podemos poner énfasis en esto.



## 8.4 Aumentar alcance

Nosotros hemos centrado el proceso en emplear la red social Instagram para extraer los datos. Pero como bien sabemos, actualmente existen además de Instagram un montón de redes sociales como Facebook, Twitter, LinkedIn, Snapchat y muchas más, de las que podemos extraer información muy parecida a la que extraemos nosotros. Lo único necesario es crear un *scraper* por cada una de ellas y amoldar el código de tal manera que se forme una estructura común. Visto que funciona correctamente, en un futuro podríamos realizar esto bien para completar la información, y también para mejorar la precisión de nuestro algoritmo.

## 8.5 Traslado a la práctica

En el último punto de esta sección, proponemos trasladar el sistema creado a un entorno real, como puede ser un comercio. Establecer unas reglas de actuación según los gustos del cliente y contrastar los resultados, para comprobar en que ámbitos se puede conseguir que su uso sea beneficioso según la métrica establecida (ingresos respecto a un comercio, por ejemplo).

## 9-Bibliografía

- [1] (s.f.). Obtenido de <http://blog.sngular.team/caso-estudio-psicologia-data-analytics-aplicado-al-analisis-personalidad-basado-modelo-big-five>
- [2] Alzate, I. I. (2019). *Definición de los procesos de producción del modelo VW216*.
- [3] Brownlee, J. (27 de Junio de 2019). *machinelearningmastery*. Obtenido de <https://machinelearningmastery.com/develop-a-deep-learning-caption-generation-model-in-python/>
- [4] Fabien, M. (5 de Abril de 2019). *Towards Data Science*. Obtenido de <https://towardsdatascience.com/a-guide-to-face-detection-in-python-3eab0f6b9fc1>
- [5] Fernandez, R. (10 de Junio de 2019). *unipython*. Obtenido de <https://unipython.com/deteccion-rostros-caras-ojos-haar-cascad/>
- [6] Lancho, C. G. (2019). *Detección de armas en vídeos mediante técnicas de Deep Learning*.
- [7] Na8. (12 de Marzo de 2018). *aprendemachinelearning*. Obtenido de <https://www.aprendemachinelearning.com/k-means-en-python-paso-a-paso/>
- [8] Ng, A. (s.f.). *Programa especializado Aprendizaje profundo*. Obtenido de Coursera: <https://es.coursera.org/specializations/deep-learning>
- [9] R, A. D. (s.f.). *ironsistem*. Obtenido de <http://ironsistem.com/tutoriales/python/deteccion-facial-en-python-y-opencv/>
- [10] Radhakrishnan, P. (29 de Septiembre de 2017). *Towards Data Science*. Obtenido de <https://towardsdatascience.com/image-captioning-in-deep-learning-9cd23fb4d8d2>
- [11] Shuang Liu, L. B. (2018). *Image Captioning Based on Deep Neural Networks*.
- [12] TensorFlow. (6 de Marzo de 2020). Obtenido de [https://www.tensorflow.org/tutorials/text/image\\_captioning](https://www.tensorflow.org/tutorials/text/image_captioning)
- [13] Arcerga, R. (Diciembre de 2016). Obtenido de <https://github.com/rarcega/instagram-scraper>